Red Hat JBoss Enterprise Application Platform 7.0

Developing EJB Applications

For Use with Red Hat JBoss Enterprise Application Platform 7.0

Last Updated: 2018-02-08

Red Hat JBoss Enterprise Application Platform 7.0 Developing EJB Applications

For Use with Red Hat JBoss Enterprise Application Platform 7.0

Legal Notice

Abstract

This document provides information for developers and administrators who want to develop and deploy EJB applications with JBoss EAP 7.0.

Table of Contents

CHAPTER 1. INTRODUCTION

1.1. OVERVIEW OF ENTERPRISE JAVABEANS

Enterprise JavaBeans (EJB) 3.2 is an API for developing distributed, transactional, secure and portable Java EE applications through the use of server-side components called Enterprise Beans. Enterprise Beans implement the business logic of an application in a decoupled manner that encourages reuse. Enterprise JavaBeans 3.2 is documented as the Java EE specification JSR-345.

EJB 3.2 provides two profiles: full and lite. JBoss EAP 7 implements the full profile for applications built using the EJB 3.2 specifications.

1.2. EJB 3.2 FEATURE SET

The following EJB 3.2 features are supported by JBoss EAP 7:

- Session beans
- Message-driven beans
- EJB API groups
- No-interface views
- Local interfaces
- Remote interfaces
- AutoClosable interface
- Timer service
- Asynchronous calls
- Interceptors
- RMI/IIOP interoperability
- Transaction support
- Security
- Embeddable API

The following features are no longer supported by JBoss EAP 7:

- EJB 2.1 entity bean client views
- Entity beans with bean-managed persistence
- Entity beans with container-managed persistence
- EJB Query Language (EJB QL)
- JAX-RPC based web services: endpoints and client views

1.3. ENTERPRISE BEANS

Enterprise beans are written as Java classes and annotated with the appropriate EJB annotations. They can be deployed to the application server in their own archive (a JAR file) or be deployed as part of a Java EE application. The application server manages the lifecycle of each enterprise bean and provides services to them such as security, transactions and concurrency management.

An enterprise bean can also define any number of business interfaces. Business interfaces provide greater control over which of the bean's methods are available to clients and can also allow access to clients running in remote JVMs.

There are three types of Enterprise beans: Session beans, Message-driven beans and Entity beans.

NOTE

JBoss EAP does not support entity beans.

1.3.1. Writing Enterprise Beans

Enterprise beans are packaged and deployed in Java archive (JAR) files. You can deploy an enterprise bean JAR file to your application server, or include it in an enterprise archive (EAR) file and deploy it with that application. You can also deploy enterprise beans in a web archive (WAR) file alongside a web application.

1.4. ENTERPRISE BEAN BUSINESS INTERFACES

An EJB business interface is a Java interface written by the bean developer which provides declarations of the public methods of a session bean that are available for clients. Session beans can implement any number of interfaces, including none (a *no-interface* bean).

Business interfaces can be declared as local or remote interfaces, but not both.

EJB Local Business Interfaces
An EJB local business interface declares the methods which are available when the bean and the client are in the same JVM. When a session bean implements a local business interface only the methods declared in that interface will be available to clients.

EJB Remote Business Interfaces
An EJB remote business interface declares the methods which are available to remote clients. Remote access to a session bean that implements a remote interface is automatically provided by the EJB container.

A remote client is any client running in a different JVM and can include desktop applications as well as web applications, services and enterprise beans deployed to a different application server.

Local clients can access the methods exposed by a remote business interface.

EJB No-interface Beans
A session bean that does not implement any business interfaces is called a no-interface bean. All of the public methods of no-interface beans are accessible to local clients.

A session bean that implements a business interface can also be written to expose a *no-interface* view.

CHAPTER 2. CREATING ENTERPRISE BEAN PROJECTS

2.1. CREATE AN EJB ARCHIVE PROJECT USING RED HAT JBOSS DEVELOPER STUDIO

This task describes how to create an Enterprise JavaBeans (EJB) project in Red Hat JBoss Developer Studio.

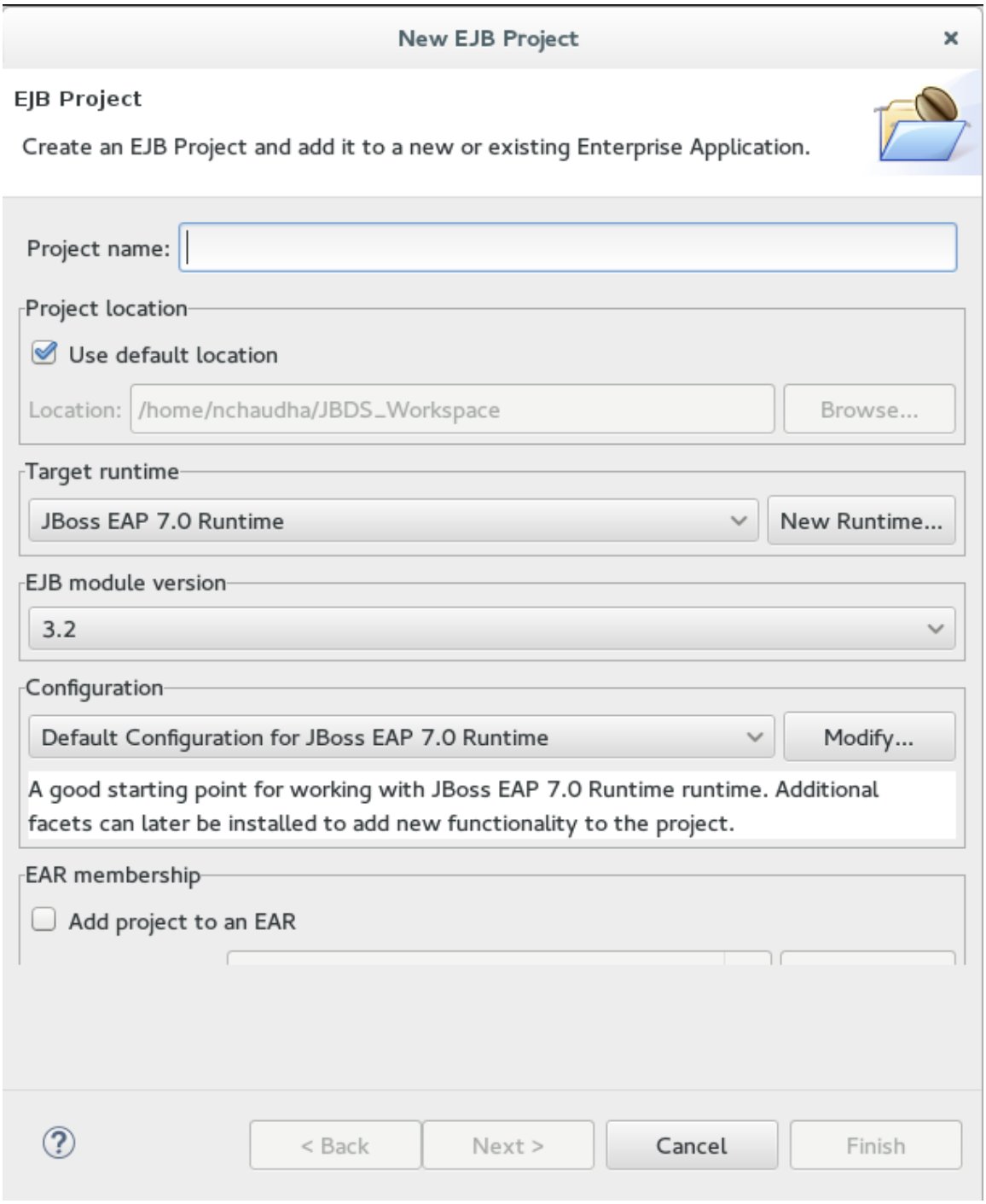
Prerequisites

- A server and server runtime for JBoss EAP has been configured in JBoss Developer Studio.

Create an EJB Project in Red Hat JBoss Developer Studio

1. Open the **New EJB Project** Wizard.
   a. Navigate to the **File** menu, select **New**, then select **Project**.
   b. When the **New Project** wizard appears, select **EJB**/**EJB Project** and click **Next**.

Figure 2.1. New EJB Project Wizard



2. Enter the following details:

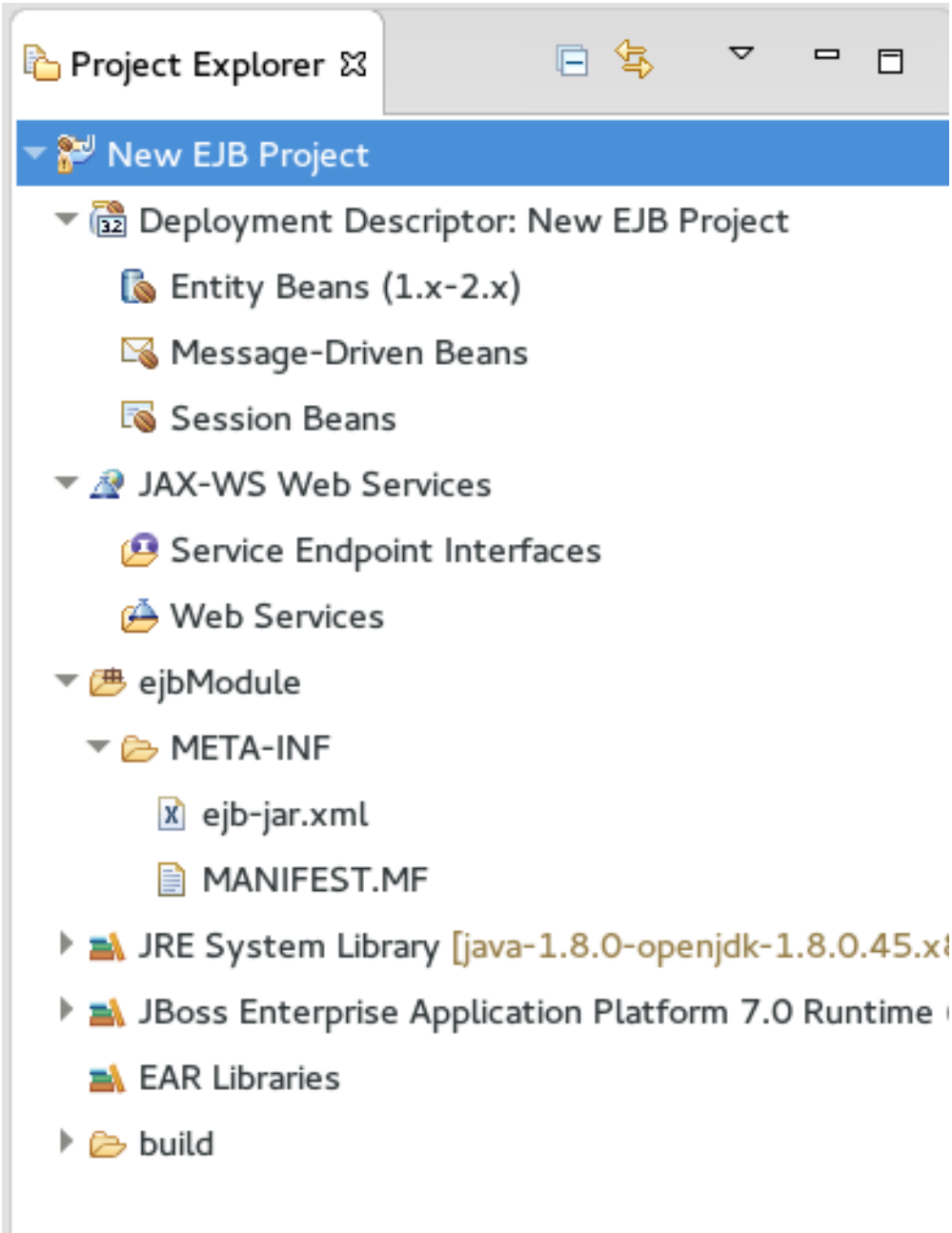
   - **Project name:** The name of the project that appears in Red Hat JBoss Developer Studio, and also the default filename for the deployed JAR file.
   - **Project location:** The directory where the project files will be saved. The default is a directory in the current workspace.
   - **Target Runtime:** This is the server runtime used for the project. This will need to be set to the same **JBoss EAP** runtime used by the server that you will be deploying to.
   - **EJB module version:** This is the version of the EJB specification that your enterprise beans will comply with. Red Hat recommends using **3.2**.

- **Configuration:** This allows you to adjust the supported features in your project. Use the default configuration for your selected runtime.
  Click **Next** to continue.

3. The **Java** project configuration screen allows you to add directories containing Java source files and specify the directory for the output of the build.
   Leave this configuration unchanged and click **Next**.

4. In the **EJB Module** settings screen, check **Generate ejb-jar.xml deployment descriptor** if a deployment descriptor is required. The deployment descriptor is optional in EJB 3.2 and can be added later if required.
   Click **Finish** and the project is created and will be displayed in the Project Explorer.

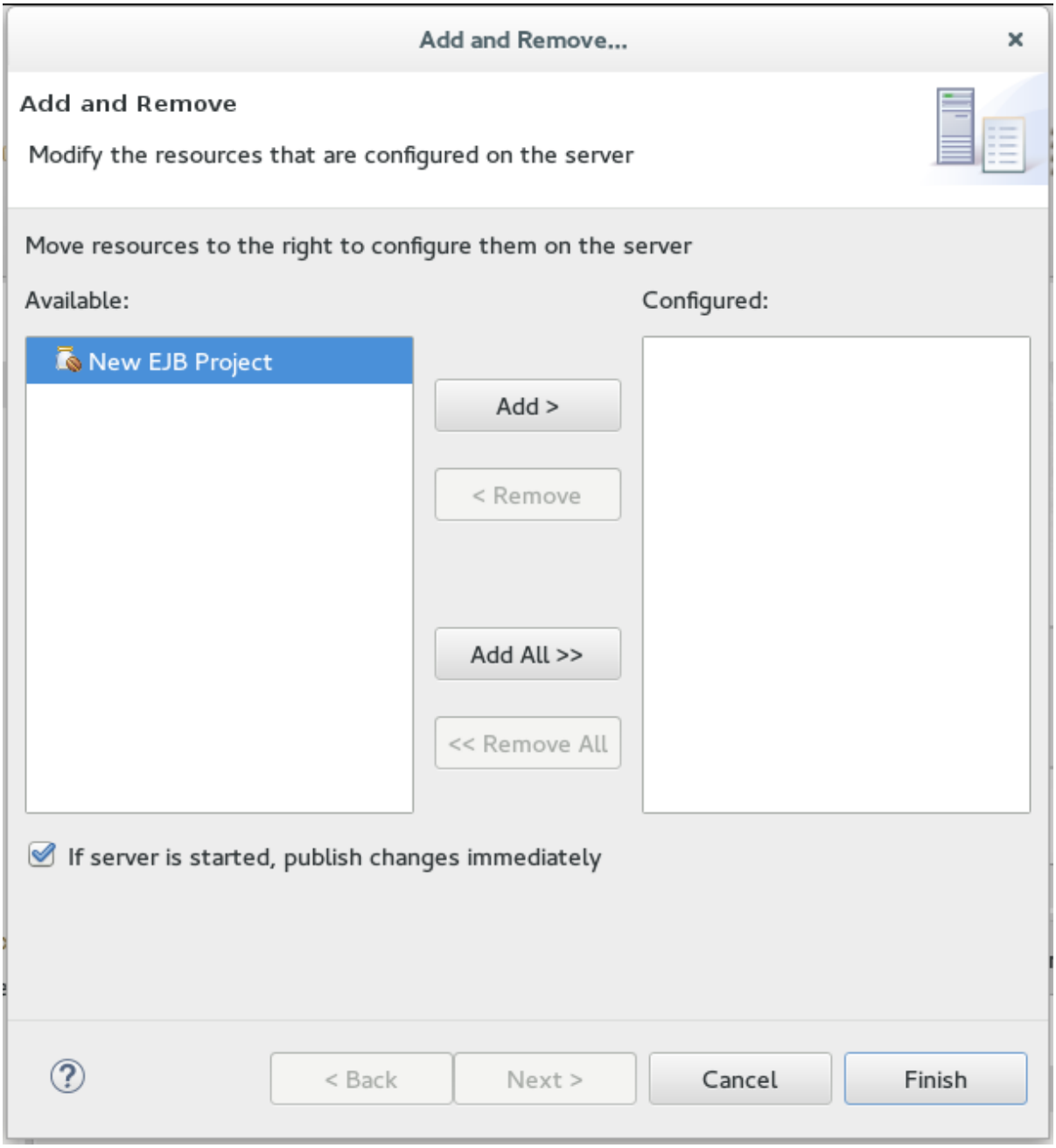
Figure 2.2. Newly created EJB Project in the Project Explorer



5. To add the project to the server for deployment, right-click on the target server in the **Server** tab and choose **Add and Remove**.

In the **Add and Remove** dialog, select the resource to deploy from the **Available** column and click the **Add** button. The resource will be moved to the **Configured** column. Click **Finish** to close the dialog.

Figure 2.3. Add and Remove dialog



You now have an EJB Project in Red Hat JBoss Developer Studio that can build and deploy to the specified server.

WARNING

If no enterprise beans are added to the project then Red Hat JBoss Developer Studio will display the warning stating ***An EJB module must contain one or more enterprise beans.*** This warning will disappear once one or more enterprise beans have been added to the project.

2.2. CREATE AN EJB ARCHIVE PROJECT IN MAVEN

This task demonstrates how to create a project using Maven that contains one or more enterprise beans packaged in a JAR file.

Prerequisites

- Maven is already installed.
- You understand the basic usage of Maven.

Create an EJB Archive project in Maven

1. **Create the Maven project:** An EJB project can be created using Maven's archetype system and the **`ejb-javaee7`** archetype. To do this run the **`mvn`** command with parameters as shown:

```
mvn archetype:generate -
DarchetypeGroupId=org.codehaus.mojo.archetypes -
DarchetypeArtifactId=ejb-javaee7
```

Maven will prompt you for the **`groupId`**, **`artifactId`**, **`version`** and **`package`** for your project.

```
[localhost]$ mvn archetype:generate -
DarchetypeGroupId=org.codehaus.mojo.archetypes -
DarchetypeArtifactId=ejb-javaee7
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------
--------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------
--------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.0:generate (default-cli) @
standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.0:generate (default-cli) @
standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.0:generate (default-cli) @
standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Archetype [org.codehaus.mojo.archetypes:ejb-javaee7:1.5]
found in catalog remote
```

```
Define value for property 'groupId': : com.shinysparkly
Define value for property 'artifactId': : payment-arrangements
Define value for property 'version':  1.0-SNAPSHOT: :
Define value for property 'package':  com.shinysparkly: :
Confirm properties configuration:
groupId: com.company
artifactId: payment-arrangements
version: 1.0-SNAPSHOT
package: com.company.collections
Y: :
[INFO] ------------------------------------------------------------
--------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------
--------------
[INFO] Total time: 32.440s
[INFO] Finished at: Mon Oct 31 10:11:12 EST 2011
[INFO] Final Memory: 7M/81M
[INFO] ------------------------------------------------------------
--------------
[localhost]$
```

2. **Add your enterprise beans:** Write your enterprise beans and add them to the project under the **`src/main/java`** directory in the appropriate sub-directory for the bean's package.

3. **Build the project:** To build the project, run the **`mvn package`** command in the same directory as the **`pom.xml`** file. This will compile the Java classes and package the JAR file. The built JAR file is named **`-.jar`** and is placed in the **`target/`** directory.

You now have a Maven project that builds and packages a JAR file. This project can contain enterprise beans and the JAR file can be deployed to an application server.

2.3. CREATE AN EAR PROJECT CONTAINING AN EJB PROJECT

This task describes how to create a new Enterprise Archive (EAR) project in Red Hat JBoss Developer Studio that contains an EJB Project.

Prerequisites

- A server and server runtime for JBoss EAP have been set up.

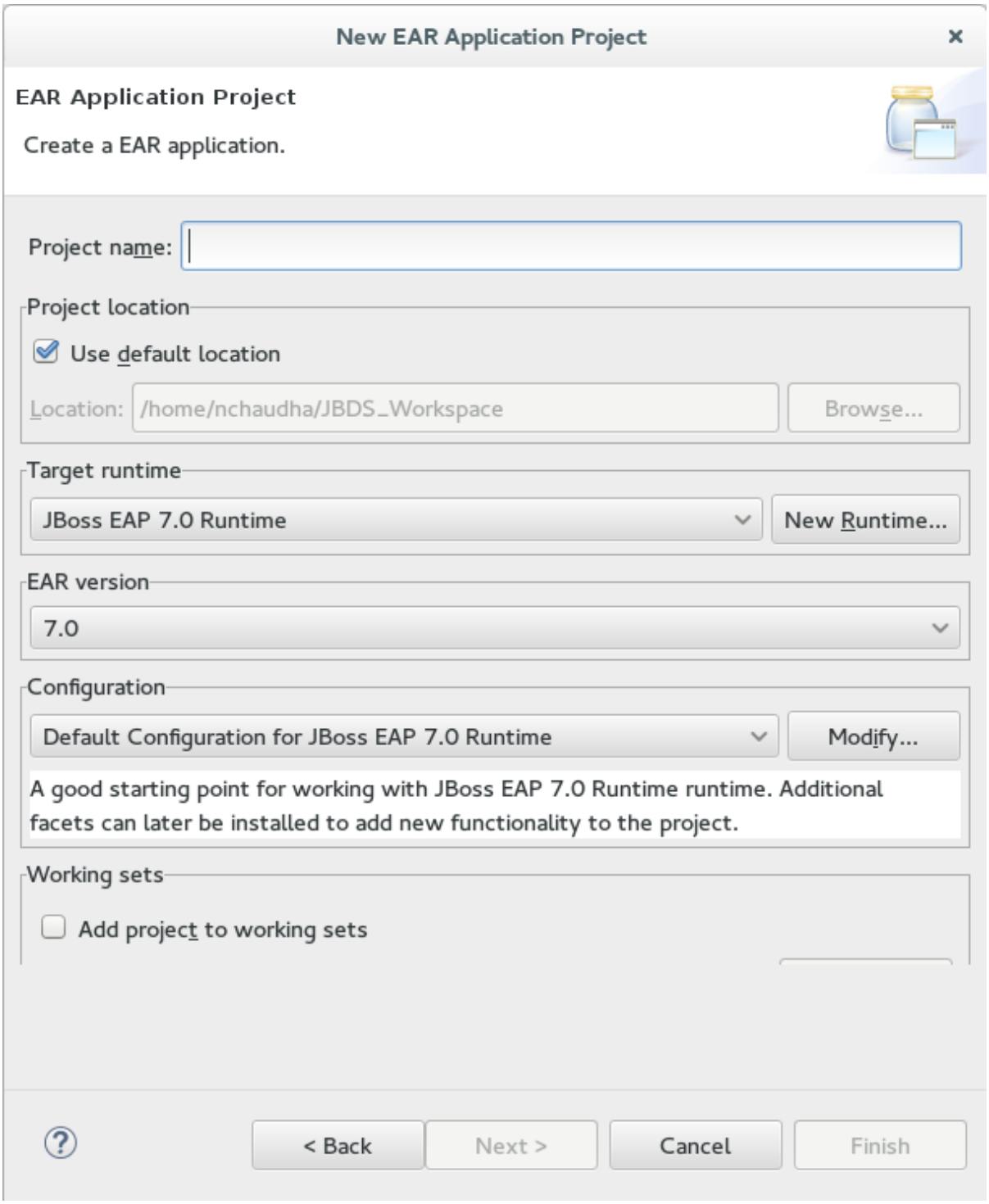
Create an EAR Project containing an EJB Project

1. Open the **New Java EE EAR Project** Wizard.

   a. Navigate to the **File** menu, select **New**, then select **Project**.

   b. When the **New Project** wizard appears, select **Java EE/Enterprise Application Project** and click **Next**.

Figure 2.4. New EAR Application Project Wizard



2. **Enter details:** Enter the following details:

   - **Project name:** The name of the project that appears in Red Hat JBoss Developer Studio, and also the default filename for the deployed EAR file.
   - **Project location:** The directory where the project files will be saved. The default is a directory in the current workspace.
   - **Target Runtime:** This is the server runtime used for the project. This will need to be set to the same JBoss EAP runtime used by the server that you will be deploying to.
   - **EAR version:** This is the version of the Java Enterprise Edition specification that your project will comply with. Red Hat recommends using Java EE 7.

- **Configuration:** This allows you to adjust the supported features in your project. Use the default configuration for your selected runtime.
  Click **Next** to continue.

3. **Add a new EJB Module:** New modules can be added from the **Enterprise Application** page of the wizard. To add a new EJB Project as a module follow the steps below:

   a. **Add new EJB Module:** Click **New Module**, uncheck **Create Default Modules** checkbox, select the **Enterprise Java Bean** and click **Next**. The **New EJB Project wizard** appears.

   b. **Create EJB Project:** New EJB Project wizard is the same as the wizard used to create new standalone EJB Projects and is described in Create an EJB Archive Project Using Red Hat JBoss Developer Studio.
   The minimal details required to create the project are:

      - Project name
      - Target Runtime
      - EJB Module version
      - Configuration
        All the other steps of the wizard are optional. Click **Finish** to complete creating the EJB Project.

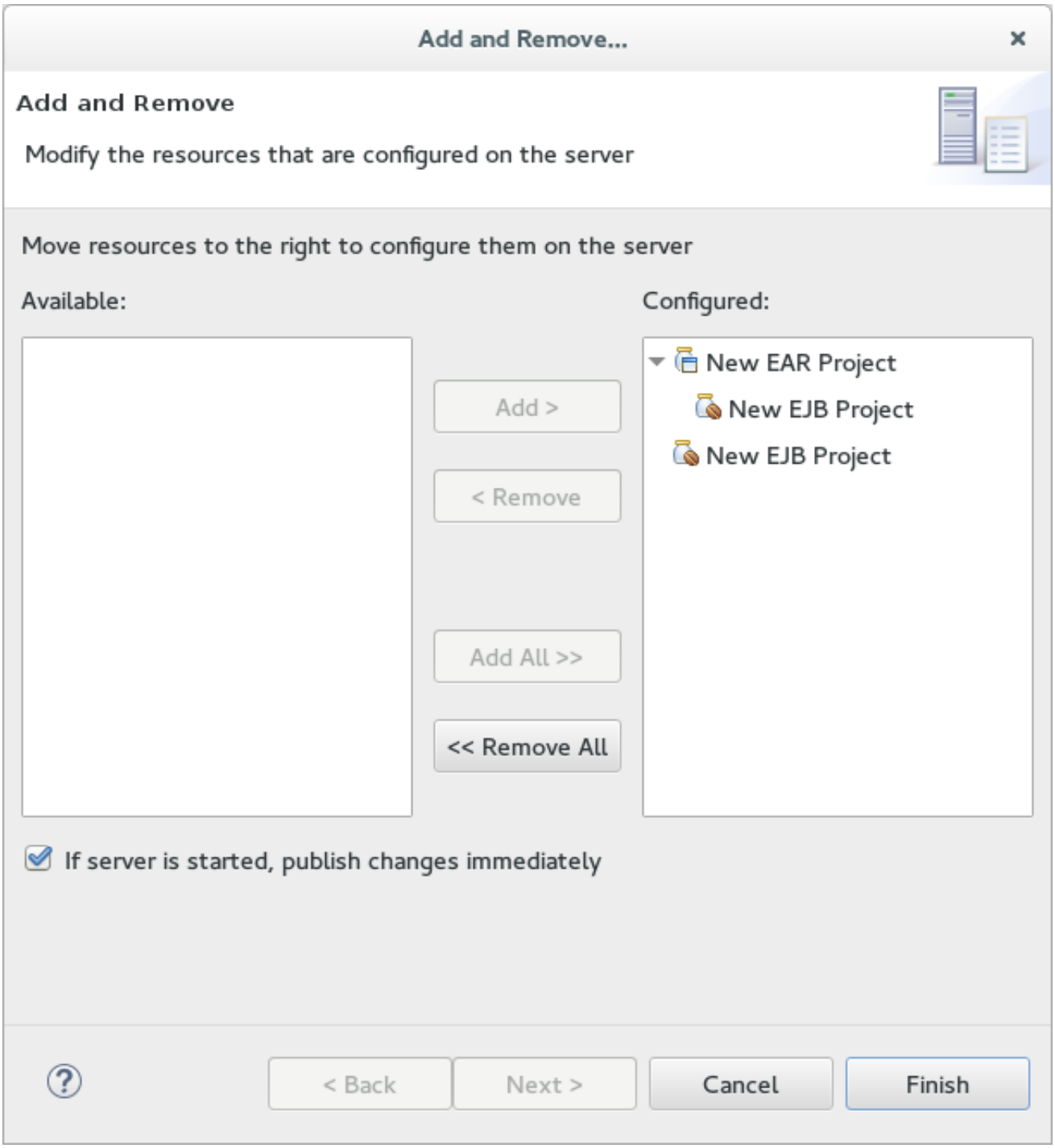
        The newly created EJB project is listed in the Java EE module dependencies and the checkbox is checked.

4. **Optional: Add an application.xml deployment descriptor:** Check the **Generate application.xml deployment descriptor** checkbox if one is required.

5. **Click Finish:** Two new projects will appear: the EJB project and the EAR project.

6. **Add Build Artifact to Server for Deployment:** Open the **Add and Remove** dialog by right-clicking in the **Servers** tab on the server you want to deploy the built artifact to in the server tab and then select **Add and Remove**.
   Select the EAR resource to deploy from the **Available** column and click the **Add** button. The resource will be moved to the Configured column. Click **Finish** to close the dialog.

Figure 2.5. Add and Remove dialog



You now have an Enterprise Application Project with a member EJB Project. This will build and deploy to the specified server as a single EAR deployment containing an EJB subdeployment.

2.4. ADD A DEPLOYMENT DESCRIPTOR TO AN EJB PROJECT

An EJB deployment descriptor can be added to an EJB project that was created without one. To do this, follow the procedure below.

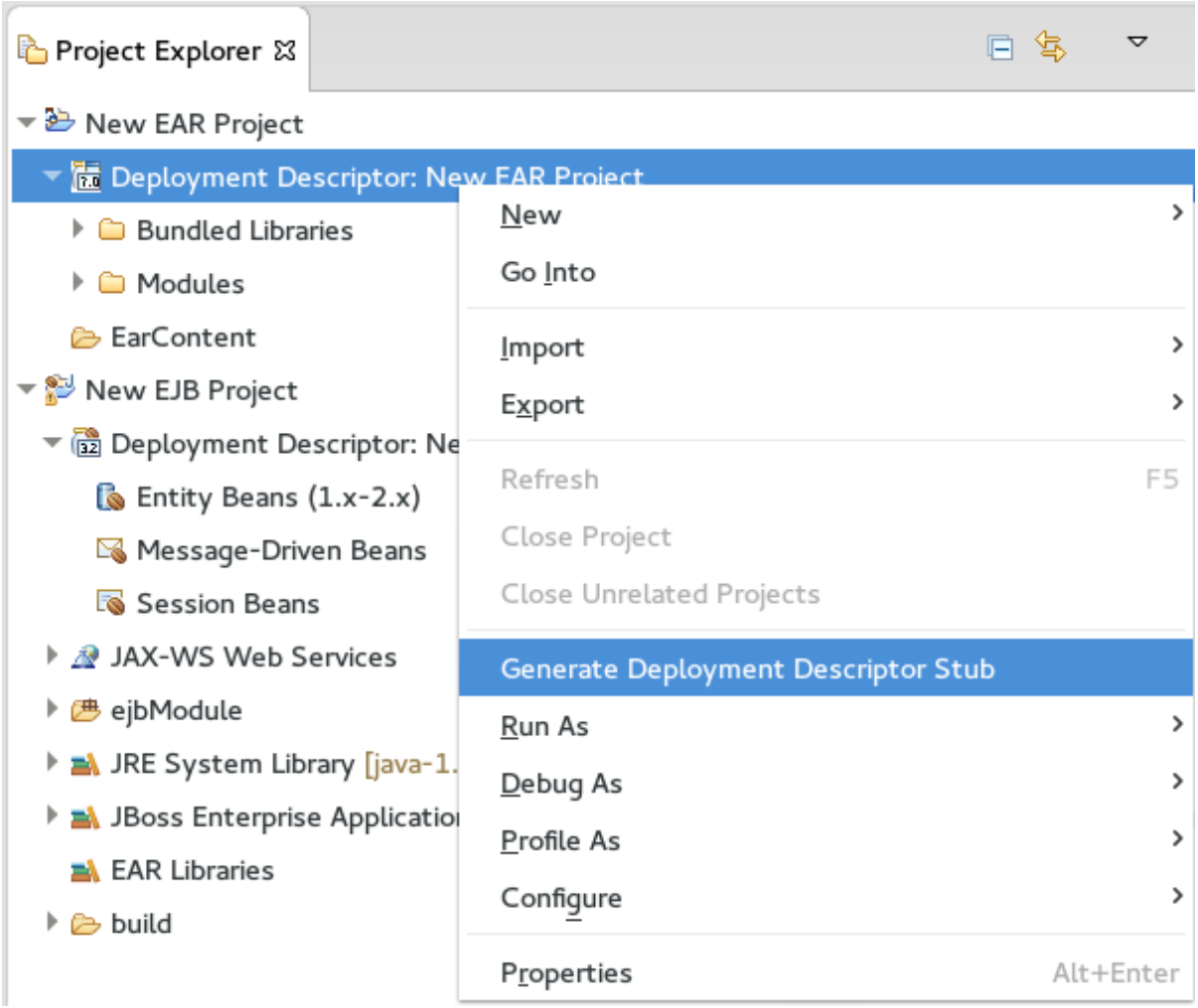
Prerequisites

- You have a EJB Project in Red Hat JBoss Developer Studio to which you want to add an EJB deployment descriptor.

Add a Deployment Descriptor to an EJB Project

1. **Open the Project:** Open the project in Red Hat JBoss Developer Studio.
2. **Add Deployment Descriptor:** Right-click on the **Deployment Descriptor** folder in the project view and select **Generate Deployment Descriptor** tab.

Figure 2.6. Adding a Deployment Descriptor



The new file, **`ejb-jar.xml`**, is created in **`ejbModule/META-INF/`**. Double-click on the **Deployment Descriptor** folder in the project view to open this file.

CHAPTER 3. SESSION BEANS

3.1. SESSION BEANS

Session Beans are Enterprise Beans that encapsulate a set of related business processes or tasks and are injected into the classes that request them. There are three types of session bean: stateless, stateful, and singleton.

3.2. STATELESS SESSION BEANS

Stateless session beans are the simplest yet most widely used type of session bean. They provide business methods to client applications but do not maintain any state between method calls. Each method is a complete task that does not rely on any shared state within that session bean. Because there is no state, the application server is not required to ensure that each method call is performed on the same instance. This makes stateless session beans very efficient and scalable.

3.3. STATEFUL SESSION BEANS

Stateful session beans are Enterprise Beans that provide business methods to client applications and maintain conversational state with the client. They should be used for tasks that must be done in several steps (method calls), each of which relies on the state of the previous step being maintained. The application server ensures that each client receives the same instance of a stateful session bean for each method call.

3.4. SINGLETON SESSION BEANS

Singleton session beans are session beans that are instantiated once per application and every client request for a singleton bean goes to the same instance. Singleton beans are an implementation of the Singleton Design Pattern as described in the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; published by Addison-Wesley in 1994.

Singleton beans provide the smallest memory footprint of all the session bean types but must be designed as thread-safe. EJB 3.2 provides container-managed concurrency (CMC) to allow developers to implement thread safe singleton beans easily. However singleton beans can also be written using traditional multi-threaded code (bean-managed concurrency or BMC) if CMC does not provide enough flexibility.

3.5. ADD SESSION BEANS TO A PROJECT IN RED HAT JBOSS DEVELOPER STUDIO

Red Hat JBoss Developer Studio has several wizards that can be used to quickly create enterprise bean classes. The following procedure shows how to use the Red Hat JBoss Developer Studio wizards to add a session bean to a project.

Prerequisites

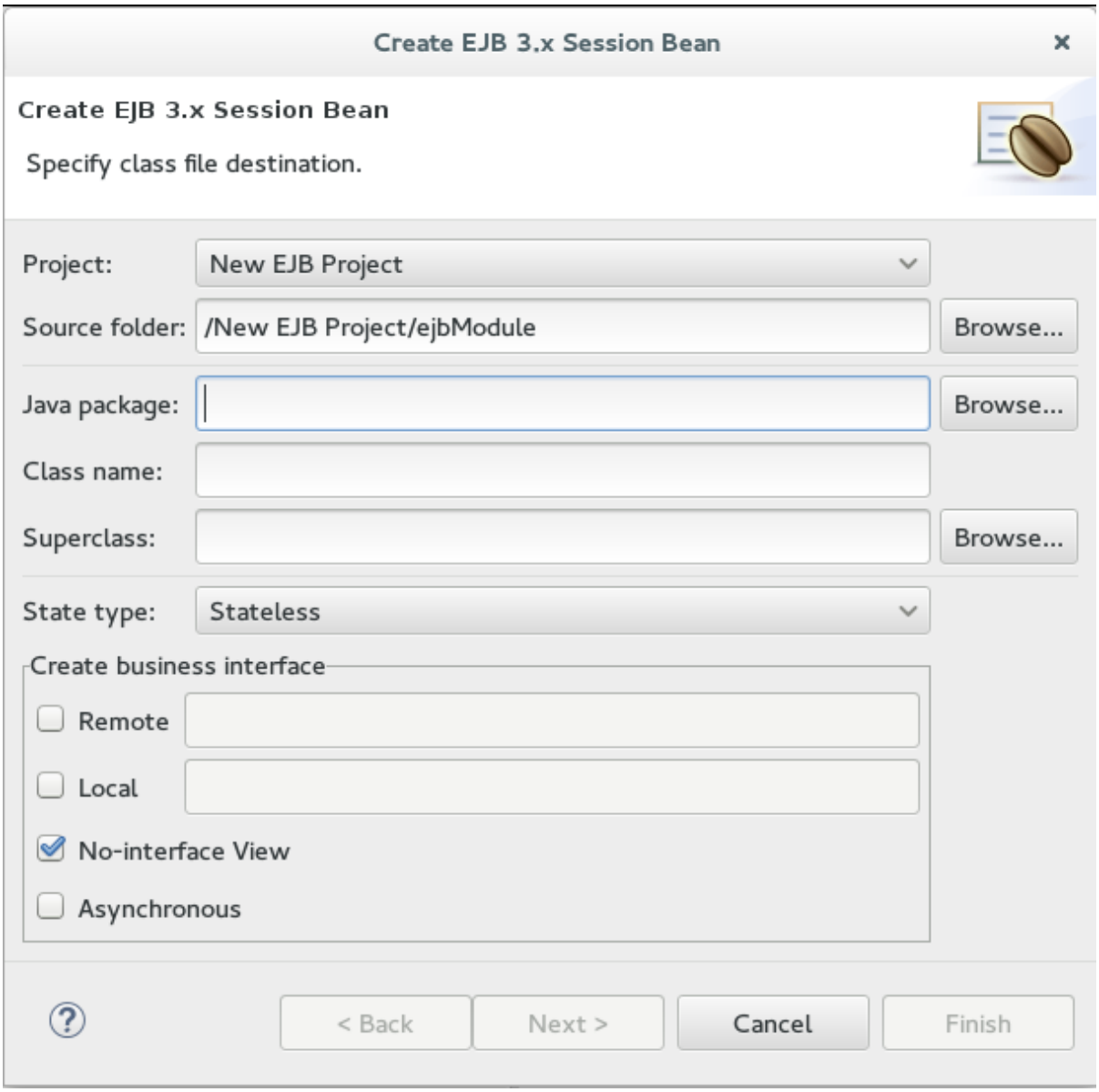
- You have a EJB or Dynamic Web Project in Red Hat JBoss Developer Studio to which you want to add one or more session beans.

Add Session Beans to a Project in Red Hat JBoss Developer Studio

1. **Open the Project:** Open the project in Red Hat JBoss Developer Studio.

2. **Open the Create EJB 3.x Session Bean wizard:** To open the **Create EJB 3.x Session Bean wizard**, navigate to the **File** menu, select **New** and then select **Session Bean (EJB 3.x)**.

Figure 3.1. Create EJB 3.x Session Bean wizard



3. **Specify class information:** Supply the following details:

- **Project:** Verify the correct project is selected.
- **Source folder:** This is the folder that the Java source files will be created in. This should not usually need to be changed.
- **Package:** Specify the package that the class belongs to.
- **Class name:** Specify the name of the class that will be the session bean.
- **Superclass:** The session bean class can inherit from a superclass. Specify that here if your session has a superclass.
- **State type:** Specify the state type of the session bean: stateless, stateful or singleton.

- **Business interfaces:** By default the **No-interface** box is checked so no interfaces will be created. Check the boxes for the interfaces you wish to define and adjust the names if necessary.
  Remember that enterprise beans in a web archive (WAR) only support EJB 3.2 Lite and this does not include remote business interfaces.

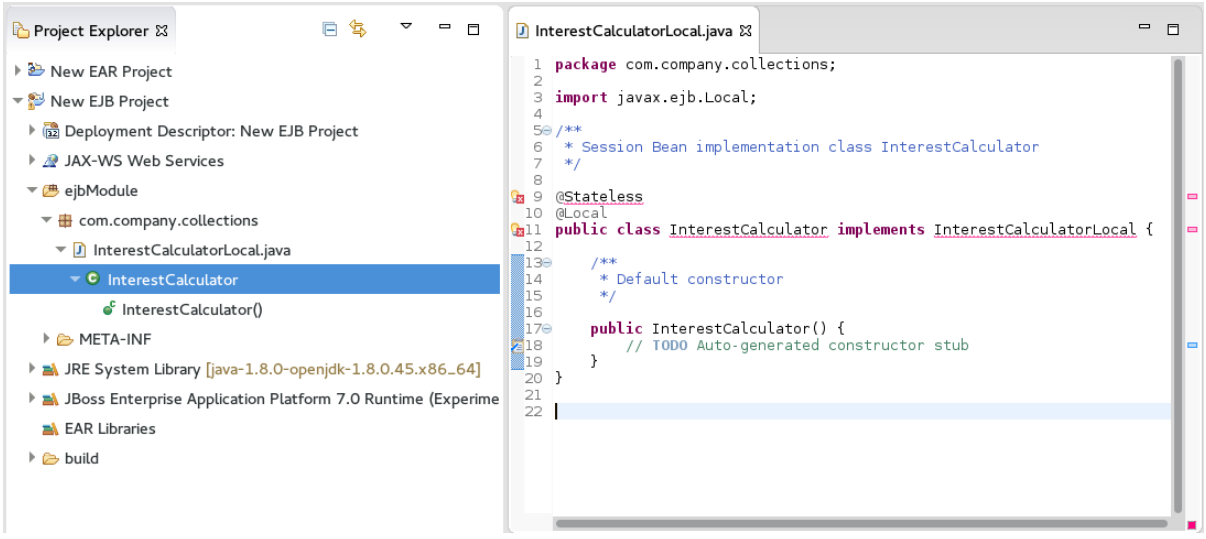
  Click **Next**.

4. **Session Bean Specific Information:** You can enter in additional information here to further customize the session bean. It is not required to change any of the information here.
   Items that you can change are:

   - Bean name
   - Mapped name
   - Transaction type (Container managed or Bean managed)
   - Additional interfaces can be supplied that the bean must implement
   - You can also specify EJB 2.x Home and Component interfaces if required

5. **Finish:** Click **Finish** and the new session bean will be created and added to the project. The files for any new business interfaces will also be created if they were specified.

Figure 3.2. New Session Bean in Red Hat JBoss Developer Studio



CHAPTER 4. MESSAGE-DRIVEN BEANS

4.1. MESSAGE-DRIVEN BEANS

Message-driven Beans (MDBs) provide an event driven model for application development. The methods of MDBs are not injected into or invoked from client code but are triggered by the receipt of messages from a messaging service such as a Java Messaging Service (JMS) server. The Java EE specification requires that JMS is supported but other messaging systems can be supported as well.

With an MDB, Java EE applications process messages asynchronously. An MDB functions as a JMS or JCA message listener. The messages can be sent by a Java EE component, for example an application client, or another enterprise bean, or by a non-Java EE application.

4.2. MESSAGE-DRIVEN BEANS CONTROLLED DELIVERY

JBoss EAP provides three attributes that control active reception of messages on a specific MDB:

- Delivery Active
- Delivery Groups
- Clustered Singleton MDBs

4.2.1. Delivery Active

The delivery active configuration of the message-driven beans (MDB) indicates whether the MDB is receiving messages or not. If an MDB is not receiving messages, then the messages will be saved in the queue or topic according to the topic or queue rules.

You can configure the **`active`** attribute of the **`delivery-group`** using XML or annotations, and you can change its value after deployment using the management CLI. By default, the **`active`** attribute is activated and delivery of messages occurs as soon as the MDB is deployed.

Configuring Delivery Active in the jboss-ejb3.xml File
In the **`jboss-ejb3.xml`** file, configure the value of **`active`** as **`false`** to indicate that the MDB will not be receiving messages as soon as it is deployed:

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:d="urn:delivery-active:1.1"
    xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-
jar_3_1.xsd"
    version="3.1"
    impl-version="2.0">
    <assembly-descriptor>
        <d:delivery>
            <ejb-name>HelloWorldQueueMDB</ejb-name>
            <d:active>false</d:active>
```

```
        </d:delivery>
    </assembly-descriptor>
</jboss:ejb-jar>
```

If you want to apply the active value to all MDBs in your application, you can use a wildcard ***** in place of the **`ejb-name`**.

Configuring Delivery Active Using Annotations
You can also use the **`org.jboss.ejb3.annotation.DeliveryActive`** annotation. For example:

```
@MessageDriven(name = "HelloWorldMDB", activationConfig = {
 @ActivationConfigProperty(propertyName = "destinationType", propertyValue
= "javax.jms.Queue"),
 @ActivationConfigProperty(propertyName = "destination", propertyValue =
"queue/HELLOWORLDMDBQueue"),
 @ActivationConfigProperty(propertyName = "acknowledgeMode", propertyValue
= "Auto-acknowledge") })
@DeliveryActive(false)

public class HelloWorldMDB implements MessageListener {
    public void onMessage(Message rcvMessage) {
      // ...
    }
}
```

If you use Maven to build your project, make sure you add the following dependency to the **`pom.xml`** file of your project:

```
<dependency>
    <groupId>org.jboss.ejb3</groupId>
    <artifactId>jboss-ejb3-ext-api</artifactId>
    <version>2.2.0.Final</version>
</dependency>
```

Configuring Delivery Active Using the Management CLI
You can configure the **`active`** attribute of the **`delivery-group`** after deployment using the management CLI. These management operations dynamically change the value of the **`active`** attribute, enabling or disabling delivery for the MDB. This method of changing the delivery active value does not persist if you restart the server. At runtime, connect to the instance you want to manage, then enter the path of the MDB for which you want to manage the delivery. For example:

- Connect to the instance you want to manage:

  ```
  cd deployment=jboss-helloworld-mdb.war/subsystem=ejb3/message-
  driven-bean=HelloWorldQueueMDB
  ```

- To stop the delivery to the MDB:

  ```
  :stop-delivery
  ```

- To start the delivery to the MDB:

  ```
  :start-delivery
  ```

View the MDB Delivery Active Status
You can view the current delivery active status of any MDB using the management console:

1. Choose **Deployments** and select the deployed MDB application. Click on **View**.
2. Expand the application and select the subsystem **`message-driven-bean`**.
3. Select the child resource, for example **`HelloWorldQueueMDB`**. Click on **View**.

Result

You see the status as **`Delivery active: true`** or **`Delivery active: false`**.

4.2.2. Delivery Groups

Delivery groups provide a way to manage the **`delivery-active`** state for a group of MDBs. Every MDB belonging to a delivery group has delivery active if and only if that group is active, and has delivery inactive whenever the group is not active.

You can add a delivery group to the **`ejb3`** subsystem using either the XML configuration or the management CLI.

Configuring Delivery Group in the jboss-ejb3.xml File

```
<delivery>
<ejb-name>MdbName<ejb-name>
<delivery-group>passive</delivery-group>
</delivery>
```

On the server side, **`delivery-groups`** can be enabled by having their **`active`** attribute set to **`true`**, or disabled by having their **`active`** attribute set to **`false`**, as shown in the example below:

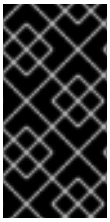
```
<delivery-groups>
<delivery-group name="group" active="true"/>
</delivery-groups>
```

Configuring Delivery Group Using the Management CLI
The state of **`delivery-groups`** can be updated using the management CLI. For example:

```
./subsystem=ejb3/mdb-delivery-group=group:add
./subsystem=ejb3/mdb-delivery-group=group:remove
./subsystem=ejb3/mdb-delivery-group=group:write-
attribute(name=active,value=true)
```

When you set the delivery active in the **`jboss-ejb3.xml`** file or using the annotation, it persists on server restart. However, when you use the management CLI to stop or start the delivery, it does not persist on server restart.

4.2.3. Clustered Singleton MDBs



IMPORTANT

This feature is provided as Technology Preview only. It is not supported for use in a production environment, and it may be subject to significant future changes. See Technology Preview Features Support Scope on the Red Hat Customer Portal for information about the support scope for Technology Preview features.

When an MDB is identified as a clustered singleton and is deployed in a cluster, only one node is active. This node can consume messages serially. When the server node fails, the active node from the clustered singleton MDBs starts consuming the messages.

Identify an MDB as a Clustered Singleton

You can use one of the following procedures to identify an MDB as a clustered singleton.

- Use the clustered-singleton XML element as shown in the example below:

```
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:c="urn:clustering:1.1"
xmlns:d="urn:delivery:1.1"
xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/ejb-jar_3_1.xsd"
version="3.1"
impl-version="2.0">
    <c:clustering>
        <ejb-name>HelloWorldQueueMDB</ejb-name>
        <c:clustered-singleton>true</c:clustered-singleton>
    </c:clustering>
</jboss:ejb-jar>
```

- In your **MDB** class, use the **@org.jboss.ejb3.annotation.ClusteredSingleton**. This procedure requires no extra configuration at the server. You need to run the service in a clustered environment.

NOTE

You have to activate the **delivery-group** in the entire cluster, specifically, in all nodes of the cluster, because you do not know which node of the cluster is chosen to be the **singleton master**. If the server chooses a node to be **singleton master**, and that node does not have the required **delivery-group** activated, no node in the cluster receives the messages.

4.3. CREATE A JMS-BASED MESSAGE-DRIVEN BEAN IN RED HAT JBOSS DEVELOPER STUDIO

This procedure shows how to add a JMS-based Message-Driven Bean to a project in Red Hat JBoss Developer Studio. This procedure creates an EJB 3.x Message-Driven Bean that uses annotations.

Prerequisites

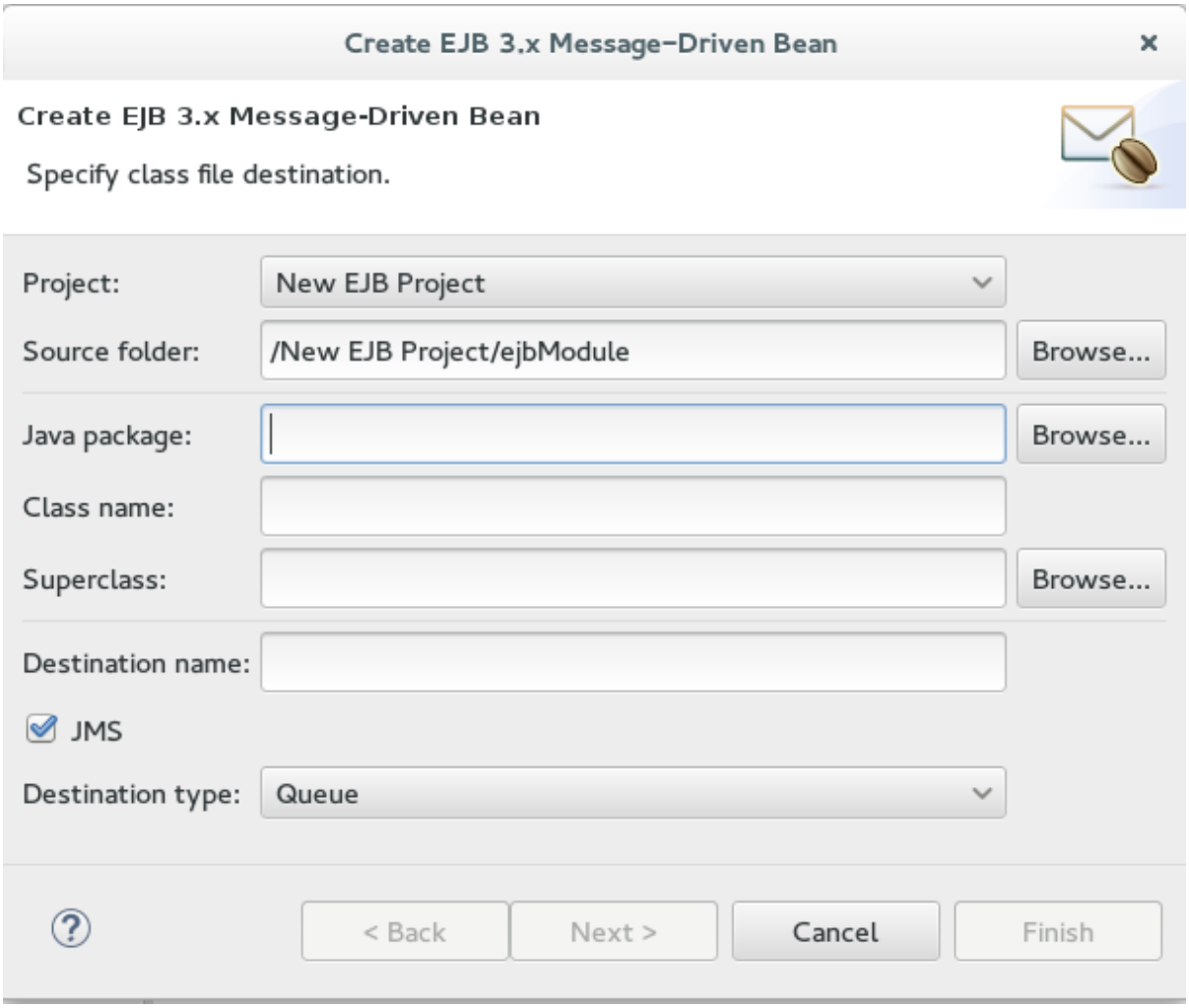
- You must have an existing project open in Red Hat JBoss Developer Studio.

- You must know the name and type of the JMS destination that the bean will be listening to.
- Support for Java Messaging Service (JMS) must be enabled in the JBoss EAP configuration to which this bean will be deployed.

Add a JMS-based Message-Driven Bean in Red Hat JBoss Developer Studio

1. **Open the Create EJB 3.x Message-Driven Bean Wizard:** Go to **File** → **New** → **Other**. Select **EJB/Message-Driven Bean (EJB 3.x)** and click the **Next** button.

   Figure 4.1. Create EJB 3.x Message-Driven Bean Wizard

2. **Specify class file destination details:** There are three sets of details to specify for the bean class here: Project, Java class, and message destination.

   - **Project:**
     - If multiple projects exist in the Workspace, ensure that the correct one is selected in the Project menu.
     - The folder where the source file for the new bean will be created is **`ejbModule`** under the selected project's directory. Only change this if you have a specific requirement.
   - **Java Class:**
     - The required fields are: Java package and class name.

- It is not necessary to supply a Superclass unless the business logic of your application requires it.

- **Message Destination:**
  - These are the details you must supply for a JMS-based Message-Driven Bean:
    - Destination name, which is the queue or topic name that contains the messages that the bean will respond to.
    - By default the JMS checkbox is selected. Do not change this.
    - Set Destination type to Queue or Topic as required.
      Click the **Next** button.

3. **Enter Message-Driven Bean specific information:** The default values here are suitable for a JMS-based Message-Driven bean using Container-managed transactions.
   - Change the **Transaction type** to Bean if the Bean will use Bean-managed transactions.
   - Change the **Bean name** if a different bean name than the class name is required.
   - The **JMS Message Listener** interface will already be listed. You do not need to add or remove any interfaces unless they are specific to your application's business logic.
   - Leave the checkboxes for creating method stubs selected.
     Click the **Finish** button.

Result

The Message-Driven Bean is created with stub methods for the default constructor and the **`onMessage()`** method. A Red Hat JBoss Developer Studio editor window opens with the corresponding file.

4.4. SPECIFYING A RESOURCE ADAPTER IN `JBOSS-EJB3.XML` FOR AN MDB

In the **`jboss-ejb3.xml`** deployment descriptor you can specify a resource adapter for an MDB to use.

To specify a resource adapter in **`jboss-ejb3.xml`** for an MDB, use the following example.

Example: jboss-ejb3.xml Configuration for an MDB Resource Adapter

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
  xmlns:jee="http://java.sun.com/xml/ns/javaee"
  xmlns:mdb="urn:resource-adapter-binding">
  <jee:assembly-descriptor>
    <mdb:resource-adapter-binding>
      <jee:ejb-name>MyMDB</jee:ejb-name>
      <mdb:resource-adapter-name>MyResourceAdapter.rar</mdb:resource-
adapter-name>
    </mdb:resource-adapter-binding>
  </jee:assembly-descriptor>
</jboss>
```

For a resource adapter located in an EAR, you must use the following syntax for **<mdb:resource-adapter-name>**:

- For a resource adapter that is in another EAR:

```
<mdb:resource-adapter-name>OtherDeployment.ear#MyResourceAdapter.rar</mdb:resource-adapter-name>
```

- For a resource adapter that is in the same EAR as the MDB, you can omit the EAR name:

```
<mdb:resource-adapter-name>#MyResourceAdapter.rar</mdb:resource-adapter-name>
```

4.5. ENABLE EJB AND MDB PROPERTY SUBSTITUTION IN AN APPLICATION

Red Hat JBoss Enterprise Application Platform allows you to enable property substitution in EJBs and MDBs using the **@ActivationConfigProperty** and **@Resource** annotations. Property substitution requires the following configuration and code changes.

- You must enable property substitution in the JBoss EAP server configuration file.
- You must define the system properties in the server configuration file or pass them as arguments when you start the JBoss EAP server.
- You must modify the application code to use the substitution variables.

The following examples demonstrate how to modify the **helloworld-mdb** quickstart that ships with JBoss EAP to use property substitution. See the **helloworld-mdb-propertysubstitution** quickstart for the completed working example.

4.5.1. Configure the Server to Enable Property Substitution

To enable property substitution in the JBoss EAP server, you must set the **<annotation-property-replacement>** attribute in the **ee** subsystem of the server configuration file to **true**.

1. Back up the server configuration file.
   The **helloworld-mdb-property-substitution** quickstart example requires the full profile for a standalone server, so this is the **standalone/configuration/standalone-full.xml** file. If you are running your server in a managed domain, this is the **domain/configuration/domain.xml** file.

2. Navigate to the JBoss EAP install directory and start the server with the full profile.

```
$ EAP_HOME/bin/standalone.sh -c standalone-full.xml
```

NOTE

For Windows Server, use the **EAP_HOME\bin\standalone.bat** script.

3. Launch the management CLI.

```
$ EAP_HOME/bin/jboss-cli.sh --connect
```

NOTE

For Windows Server, use the **`EAP_HOME\bin\jboss-cli.bat`** script.

4. Type the following command to enable annotation property substitution.

```
/subsystem=ee:write-attribute(name=annotation-property-
replacement,value=true)
```

You should see the following result.

```
{"outcome" => "success"}
```

5. Review the changes to the JBoss EAP server configuration file. The **ee** subsystem should now contain the following XML.

Example ee Subsystem Configuration

```
<subsystem xmlns="urn:jboss:domain:ee:4.0">
  ...
  <annotation-property-replacement>true</annotation-property-
replacement>
  ...
</subsystem>
```

4.5.2. Define the System Properties

You can specify the system properties in the server configuration file or you can pass them as command line arguments when you start the JBoss EAP server. System properties defined in the server configuration file take precedence over those passed on the command line when you start the server.

4.5.2.1. Define the System Properties in the Server Configuration File

1. Launch the management CLI as described above.
2. Use the following command syntax to configure a system property in the JBoss EAP server.

Syntax to Add a System Property

```
/system-property=PROPERTY_NAME:add(value=PROPERTY_VALUE)
```

The following system properties are configured for the **`helloworld-mdb-propertysubstitution`** quickstart.

Example Commands to Add System Properties

```
/system-
property=property.helloworldmdb.queue:add(value=java:/queue/HELLOWOR
LDMDBPropQueue)
```

```
/system-
property=property.helloworldmdb.topic:add(value=java:/topic/HELLOWOR
LDMDBPropTopic)
/system-
property=property.connection.factory:add(value=java:/ConnectionFacto
ry)
```

3. Review the changes to the JBoss EAP server configuration file. The following system properties should now appear in the after the **<extensions>**.

Example System Properties Configuration

```
<system-properties>
    <property name="property.helloworldmdb.queue"
value="java:/queue/HELLOWORLDMDBPropQueue"/>
    <property name="property.helloworldmdb.topic"
value="java:/topic/HELLOWORLDMDBPropTopic"/>
    <property name="property.connection.factory"
value="java:/ConnectionFactory"/>
</system-properties>
```

4.5.2.2. Pass the System Properties as Arguments on Server Start

If you prefer, you can instead pass the arguments on the command line when you start the JBoss EAP server in the form of **-DPROPERTY_NAME=PROPERTY_VALUE**. The following is an example of how to pass the arguments for the system properties defined in the previous section.

Example Server Start Command Passing System Properties

```
EAP_HOME/bin/standalone.sh -c standalone-full.xml  -
Dproperty.helloworldmdb.queue=java:/queue/HELLOWORLDMDBPropQueue  -
Dproperty.helloworldmdb.topic=java:/topic/HELLOWORLDMDBPropTopic  -
Dproperty.connection.factory=java:/ConnectionFactory
```

4.5.3. Modify the Application Code to Use the System Property Substitutions

Replace the hard-coded **@ActivationConfigProperty** and **@Resource** annotation values with substitutions for the newly defined system properties. The following are examples of how to change the **helloworld-mdb** quickstart to use the newly defined system property substitutions.

1. Change the **@ActivationConfigProperty destination** property value in the **HelloWorldQueueMDB** class to use the substitution for the system property. The **@MessageDriven** annotation should now look like this:

HelloWorldQueueMDB Code Example

```
@MessageDriven(name = "HelloWorldQueueMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "${property.helloworldmdb.queue}"),
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
propertyValue = "Auto-acknowledge") })
```

2. Change the **@ActivationConfigProperty destination** property value in the **HelloWorldTopicMDB** class to use the substitution for the system property. The **@MessageDriven** annotation should now look like this:

HelloWorldTopicMDB Code Example

```
@MessageDriven(name = "HelloWorldQTopicMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "${property.helloworldmdb.topic}"),
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Topic"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
propertyValue = "Auto-acknowledge") })
```

3. Change the **@Resource** annotations in the **HelloWorldMDBServletClient** class to use the system property substitutions. The code should now look like this:

HelloWorldMDBServletClient Code Example

```
/**
 * Definition of the two JMS destinations used by the quickstart
 * (one queue and one topic).
 */
 @JMSDestinationDefinitions(
     value = {
         @JMSDestinationDefinition(
             name = "java:/${property.helloworldmdb.queue}",
             interfaceName = "javax.jms.Queue",
             destinationName = "HelloWorldMDBQueue"
         ),
         @JMSDestinationDefinition(
             name = "java:/${property.helloworldmdb.topic}",
             interfaceName = "javax.jms.Topic",
             destinationName = "HelloWorldMDBTopic"
         )
     })
/**
 * <p>
 * A simple servlet 3 as client that sends several messages to a
queue or a topic.
 * </p>
 *
 * <p>
 * The servlet is registered and mapped to
/HelloWorldMDBServletClient using the {@linkplain WebServlet
 * @HttpServlet}.
 * </p>
 *
 * @author Serge Pagop (spagop@redhat.com)
 *
 */
@WebServlet("/HelloWorldMDBServletClient")
public class HelloWorldMDBServletClient extends HttpServlet {

    private static final long serialVersionUID = -
```

```
8314035702649252239L;

    private static final int MSG_COUNT = 5;

    @Inject
    private JMSContext context;

    @Resource(lookup = "${property.helloworldmdb.queue}")
    private Queue queue;

    @Resource(lookup = "${property.helloworldmdb.topic}")
    private Topic topic;

  <!-- Remainder of code can be found in the `helloworld-mdb-
propertysubstitution` quickstart. -->
```

4. Modify the **activemq-jms.xml** file to use the system property substitution values.

   Example .activemq-jms.xml File

```
<?xml version="1.0" encoding="UTF-8"?>
<messaging-deployment xmlns="urn:jboss:messaging-activemq-
deployment:1.0">
    <server>
         <jms-destinations>
            <jms-queue name="HELLOWORLDMDBQueue">
                <entry name="${property.helloworldmdb.queue}"/>
            </jms-queue>
            <jms-topic name="HELLOWORLDMDBTopic">
                <entry name="${property.helloworldmdb.topic}"/>
            </jms-topic>
        </jms-destinations>
    </server>
</messaging-deployment>
```

5. Deploy the application. The application now uses the values specified by the system properties for the **@Resource** and **@ActivationConfigProperty** property values.

4.6. ACTIVATION CONFIGURATION PROPERTIES

4.6.1. Configuring MDBs Using Annotations

You can configure activation properties by using the **@MessageDriven** element and sub-elements which correspond to the **@ActivationConfigProperty** annotation. **@ActivationConfigProperty** is an array of activation configuration properties for MDBs. The **@ActivationConfigProperty** annotation specification is as follows:

```
@Target(value={})
@Retention(value=RUNTIME)
public @interface ActivationConfigProperty
{
 String propertyName();
 String propertyValue();
}
```

Example showing `@ActivationConfigProperty`

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

@ActivationConfigProperty(propertyName="destinationLookup",propertyValue="
queueA"),
    @ActivationConfigProperty(propertyName =
"destinationType",propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
propertyValue = "Auto-acknowledge"),
})
```

4.6.2. Configuring MDBs Using Deployment Descriptor

The **`<message-driven>`** element in the **`ejb-jar.xml`** defines the bean as an MDB. The **`<activation-config>`** and elements contain the MDB configuration via the **`activation-config-property`** elements.

Example `ejb-jar.xml`

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
               xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/ejb-
jar_3_1.xsd"
               version="3.1">
    <enterprise-beans>
        <message-driven>
            <ejb-name>MyMDBName</ejb-name>
            <ejb-
class>org.jboss.tutorial.mdb_deployment_descriptor.bean.MyMDBName</ejb-
class>
            <activation-config>
                <activation-config-property>
                    <activation-config-property-
name>destinationLookup</activation-config-property-name>
                    <activation-config-property-value>queueA</activation-
config-property-value>
                </activation-config-property>
                <activation-config-property>
                    <activation-config-property-
name>destinationType</activation-config-property-name>
                    <activation-config-property-
value>javax.jms.Queue</activation-config-property-value>
                </activation-config-property>
                <activation-config-property>
                    <activation-config-property-
name>acknowledgeMode</activation-config-property-name>
```

```
                    <activation-config-property-value>Auto-
acknowledge</activation-config-property-value>
                </activation-config-property>
            </activation-config>
        </message-driven>
    <enterprise-beans>
</jboss:ejb-jar>
```

Table 4.1. Activation Configuration Properties defined by JMS Specifications

Name	Description
destinationLookup	The JNDI name of the queue or topic. This is a mandatory value.
connectionFactoryLookup	The lookup name of an administratively defined **javax.jms.ConnectionFactory**, **javax.jms.QueueConnectionFactory** or **javax.jms.TopicConnectionFactory** object that will be used to connect to the JMS provider from which the endpoint would receive messages. If not defined explicitly, pooled connection factory with name **activemq-ra** is used.
destinationType	The type of destination valid values are **javax.jms.Queue** or **javax.jms.Topic**. This is a mandatory value.
messageSelector	The value for a **messageSelector** property is a string which is used to select a subset of the available messages. Its syntax is based on a subset of the SQL 92 conditional expression syntax and is described in detail in JMS specification. Specifying a value for the **messageSelector** property on the **ActivationSpec** JavaBean is optional.
acknowledgeMode	The type of acknowledgement when not using transacted JMS. Valid values are **Auto-acknowledge** or **Dups-ok-acknowledge**. This is not a mandatory value. The default value is **Auto-acknowledge**.
clientID	The client ID of the connection. This is not a mandatory value.
subscriptionDurability	Whether topic subscriptions are durable. Valid values are **Durable** or **NonDurable**. This is not a mandatory value. The default value is **NonDurable**.
subscriptionName	The subscription name of the topic subscription. This is not a mandatory value.

Table 4.2. Activation Configuration Properties defined by JBoss EAP

Name	Description
destination	Using this property with **useJNDI=true** has the same meaning as **destinationLookup**. Using it with **useJNDI=false**, the destination is not looked up, but it is instantiated. You can use this property instead of **destinationLookup**. This is not a mandatory value.
shareSubscriptions	Whether the connection is configured to share subscriptions. The default value is **False**.
user	The user for the JMS connection. This is not a mandatory value.
password	The password for the JMS connection. This is not a mandatory value.
maxSession	The maximum number of concurrent sessions to use. This is not a mandatory value. The default value is **15**.
transactionTimeout	The transaction timeout for the session in milliseconds. This is not a mandatory value. If not specified or 0, the property is ignored and the **transactionTimeout** is not overridden and the default **transactionTimeout** defined in the Transaction Manager is used.
useJNDI	Whether or not use JNDI to look up the destination. The default value is **True**.
jndiParams	The JNDI parameters to use in the connection. Parameters are defined as **name=value** pairs separated by **;**
localTx	Use local transaction instead of XA. The default value is **False**.
setupAttempts	Number of attempts to set up a JMS connection. It is possible that the MDB is deployed before the JMS resources are available. In that case, the resource adapter will try to set up several times until the resources are available. This applies only to inbound connections. The default value is **-1**.
setupInterval	Interval in milliseconds between consecutive attempts to setup a JMS connection. This applies only to inbound connections. The default value is **2000**.

Name	Description
rebalanceConnections	Whether rebalancing of inbound connections is enabled or not. The default value is **False**.

4.6.3. Some Example Use Cases for Configuring MDBs

- Use case for MDB receiving a message
  For a basic scenario when MDB receives a message, see the **helloworld-mdb** quickstart that is shipped with JBoss EAP.

- Use case for MDB sending a message
  After processing the message you may need to inform other business systems or reply to the message. In this case, you can send the message from MDB as shown in the snippet below:

```
package org.jboss.as.quickstarts.mdb;

import javax.annotation.Resource;
import javax.ejb.ActivationConfigProperty;
import javax.ejb.MessageDriven;
import javax.inject.Inject;
import javax.jms.JMSContext;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.Queue;

@MessageDriven(name = "MyMDB", activationConfig = {
    @ActivationConfigProperty(propertyName = "destinationLookup",
propertyValue = "queue/MyMDBRequest"),
    @ActivationConfigProperty(propertyName = "destinationType",
propertyValue = "javax.jms.Queue"),
    @ActivationConfigProperty(propertyName = "acknowledgeMode",
propertyValue = "Auto-acknowledge") })
public class MyMDB implements MessageListener {

    @Inject
    private JMSContext jmsContext;

    @Resource(lookup = "java:/queue/ResponseDefault")
    private Queue defaultDestination;

    /**
     * @see MessageListener#onMessage(Message)
     */
    public void onMessage(Message rcvMessage) {
        try {
            Message response =
jmsContext.createTextMessage("Response for message " +
rcvMessage.getJMSMessageID());
            if (rcvMessage.getJMSReplyTo() != null) {

jmsContext.createProducer().send(rcvMessage.getJMSReplyTo(),
```

```
response);
            } else {

jmsContext.createProducer().send(defaultDestination, response);
            }
        } catch (JMSException e) {
            throw new RuntimeException(e);
        }
    }
}
```

In the example above, after the MDB receives the message, it replies to either the destination specified in **`JMSReplyTo`** or the destination which is bound to the JNDI name **`java:/queue/ResponseDefault`**.

- Use case for MDB configuring rebalancing of inbound connection

```
@MessageDriven(name="MyMDBName",
    activationConfig =
    {
        @ActivationConfigProperty(propertyName =
"destinationType",propertyValue = "javax.jms.Queue"),
        @ActivationConfigProperty(propertyName =
"destinationLookup", propertyValue = "queueA"),
        @ActivationConfigProperty(propertyName =
"rebalanceConnections", propertyValue = "true")
    }
)
```

CHAPTER 5. INVOKING SESSION BEANS

5.1. INVOKE A SESSION BEAN REMOTELY USING JNDI

This task describes how to add support to a remote client for the invocation of session beans using JNDI. The task assumes that the project is being built using Maven.

The **`ejb-remote`** quickstart contains working Maven projects that demonstrate this functionality. The quickstart contains projects for both the session beans to deploy and the remote client. The code samples below are taken from the remote client project.

This task assumes that the session beans do not require authentication.

WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

Prerequisites

- You must already have a Maven project created ready to use.
- Configuration for the JBoss EAP Maven repository has already been added.
- The session beans that you want to invoke are already deployed.
- The deployed session beans implement remote business interfaces.
- The remote business interfaces of the session beans are available as a Maven dependency. If the remote business interfaces are only available as a JAR file then it is recommended to add the JAR to your Maven repository as an artifact. See the Maven documentation for the **`install:install-file`** goal for directions, http://maven.apache.org/plugins/maven-install-plugin/usage.html
- You need to know the host name and JNDI port of the server hosting the session beans.

To invoke a session bean from a remote client you must first configure the project correctly.

Add Maven Project Configuration for Remote Invocation of Session Beans

1. Add the required project dependencies.
   The **`pom.xml`** for the project must be updated to include the necessary dependencies.

2. Add the **`jboss-ejb-client.properties`** file.
   The JBoss EJB client API expects to find a file in the root of the project named **`jboss-ejb-client.properties`** that contains the connection information for the JNDI service. Add this file to the **`src/main/resources/`** directory of your project with the following content.

   EJB Client Properties File Example

   ■

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLE
D=false
remote.connections=default
remote.connection.default.host=localhost
remote.connection.default.port = 8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLI
CY_NOANONYMOUS=false
#remote.connection.default.connect.options.org.xnio.Options.SSL_STAR
TTLS=true
```

Change the host name and port to match your server. The default port number is **8080**. For a secure connection, set the **SSL_ENABLED** line to **true** and uncomment the **SSL_STARTTLS** line. The Remoting interface in the container supports secured and unsecured connections using the same port.

3. Add dependencies on the remote business interfaces.
   Add the Maven dependencies on the remote business interfaces of the session beans to the **pom.xml**.

   POM File Configuration Example

   ```
   <dependency>
       <groupId>org.jboss.quickstarts.eap</groupId>
       <artifactId>jboss-ejb-remote-server-side</artifactId>
       <type>ejb-client</type>
       <version>${project.version}</version>
   </dependency>
   ```

   After the project is configured correctly, you can add the code to access and invoke the session beans.

Obtain a Bean Proxy using JNDI and Invoke Methods of the Bean

1. Handle checked exceptions.
   Two of the methods used in the following code (**InitialContext()** and **lookup()**) have a checked exception of type **javax.naming.NamingException**. These method calls must either be enclosed in a try/catch block that catches **NamingException** or in a method that is declared to throw **NamingException**. The **ejb-remote** quickstart uses the second technique.

2. Create a JNDI context.
   A JNDI Context object provides the mechanism for requesting resources from the server. Create a JNDI context using the following code:

   Create a JNDI Context Code Example

   ```
   final Hashtable jndiProperties = new Hashtable();
   jndiProperties.put(Context.URL_PKG_PREFIXES,
   "org.jboss.ejb.client.naming");
   final Context context = new InitialContext(jndiProperties);
   ```

   The connection properties for the JNDI service are read from the **jboss-ejb-client.properties** file.

3. Use the JNDI Context's lookup() method to obtain a bean proxy.

Invoke the **`lookup()`** method of the bean proxy and pass it the JNDI name of the session bean you require. This will return an object that must be cast to the type of the remote business interface that contains the methods you want to invoke.

Invoke Lookup Method Code Example

```
final RemoteCalculator statelessRemoteCalculator =
(RemoteCalculator) context.lookup(
    "ejb:/jboss-ejb-remote-server-side//CalculatorBean!" +
    RemoteCalculator.class.getName());
```

Session bean JNDI names are defined using a special syntax. For more information, see EJB JNDI Naming Reference.

4. **Invoke methods:** Now that you have a proxy bean object you can invoke any of the methods contained in the remote business interface.

Invoke Remote Method Code Example

```
int a = 204;
int b = 340;
System.out.println("Adding " + a + " and " + b + " via the remote
stateless calculator deployed on the server");
int sum = statelessRemoteCalculator.add(a, b);
System.out.println("Remote calculator returned sum = " + sum);
```

The proxy bean passes the method invocation request to the session bean on the server, where it is executed. The result is returned to the proxy bean which then returns it to the caller. The communication between the proxy bean and the remote session bean is transparent to the caller.

You should now be able to configure a Maven project to support invoking session beans on a remote server and write the code invoke the session beans' methods using a proxy bean retrieved from the server using JNDI.

5.2. ABOUT EJB CLIENT CONTEXTS

JBoss EAP introduced the EJB client API for managing remote EJB invocations. The JBoss EJB client API uses the EJBClientContext, which may be associated with and be used by one or more threads concurrently. The means an EJBClientContext can potentially contain any number of EJB receivers. An EJB receiver is a component that knows how to communicate with a server that is capable of handling the EJB invocation. Typically, EJB remote applications can be classified into the following:

- A remote client, which runs as a standalone Java application.
- A remote client, which runs within another JBoss EAP instance.

Depending on the type of remote client, from an EJB client API point of view, there can potentially be more than one EJBClientContext within a JVM.

While standalone applications typically have a single EJBClientContext that may be backed by any number of EJB receivers, this isn't mandatory. If a standalone application has more than one EJBClientContext, an EJB client context selector is responsible for returning the appropriate context.

In case of remote clients that run within another JBoss EAP instance, each deployed application will have a corresponding EJB client context. Whenever that application invokes another EJB, the

corresponding EJB client context is used to find the correct EJB receiver, which then handles the invocation.

5.3. CONSIDERATIONS WHEN USING A SINGLE EJB CONTEXT

Summary

You must consider your application requirements when using a single EJB client context with standalone remote clients. For more information about the different types of remote clients, refer to: About EJB Client Contexts.

Typical Process for a Remote Standalone Client with a Single EJB Client Context

A remote standalone client typically has just one EJB client context backed by any number of EJB receivers. The following is an example of a standalone remote client application:

```
public class MyApplication {
    public static void main(String args[]) {
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext();
        final MyBeanInterface beanOne =
ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
    }
}
```

Remote client JNDI lookups are usually backed by a **`jboss-ejb-client.properties`** file, which is used to set up the EJB client context and the EJB receivers. This configuration also includes the security credentials, which are then used to create the EJB receiver that connects to the JBoss EAP server. When the above code is invoked, the EJB client API looks for the EJB client context, which is then used to select the EJB receiver that will receive and process the EJB invocation request. In this case, there is just the single EJB client context, so that context is used by the above code to invoke the bean. The procedure to invoke a session bean remotely using JNDI is described in greater detail here: Invoke a Session Bean Remotely using JNDI.

Remote Standalone Client Requiring Different Credentials

A user application may want to invoke a bean more than once, but connect to the JBoss EAP server using different security credentials. The following is an example of a standalone remote client application that invokes the same bean twice:

```
public class MyApplication {
    public static void main(String args[]) {
        // Use the "foo" security credential connect to the server and
invoke this bean instance
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext();
        final MyBeanInterface beanOne =
ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...

        // Use the "bar" security credential to connect to the server and
invoke this bean instance
        final javax.naming.Context ctxTwo = new
```

```
javax.naming.InitialContext();
        final MyBeanInterface beanTwo =
ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
    }
}
```

In this case, the application wants to connect to the same server instance to invoke the EJB hosted on that server, but wants to use two different credentials while connecting to the server. Because the client application has a single EJB client context, which can have only one EJB receiver for each server instance, this means the above code uses just one credential to connect to the server and the code does not execute as the application expects it to.

Solution

Scoped EJB client contexts offer a solution to this issue. They provide a way to have more control over the EJB client contexts and their associated JNDI contexts, which are typically used for EJB invocations. For more information about scoped EJB client contexts, refer to Using Scoped EJB Client Contexts and Configure EJBs Using a Scoped EJB Client Context.

5.4. TRANSACTION BEHAVIOR OF EJB INVOCATIONS

Server to Server Invocations

Transaction attributes for distributed JBoss EAP applications must be handled such that the application is called on the same server. To discontinue a transaction, the destination method must be marked **`REQUIRES_NEW`** using different interfaces.

An EJB can be invoked using either of the following methods:

- EJB Remoting Call
- Internet Inter-ORB Protocol (IIOP) Remote Call

NOTE

JBoss EAP does not require Java Transaction Services (JTS) for transaction propagation on server-to-server EJB invocations if both servers are JBoss EAP. JBoss EJB client API library handles it itself.

EJB Remoting Call

To invoke EJB session beans with a JBoss EAP standalone client, the client must have a reference to the **`InitialContext`** object while the EJB proxies or **`UserTransaction`** are used. It is also important to keep the **`InitialContext`** object open while EJB proxies or **`UserTransaction`** are being used. Control of the connections will be inside the classes created by the **`InitialContext`** with the properties.

The following code example shows an EJB client that holds a reference to the **`InitialContext`** object. This code example was taken from the **`ejb-multi-server`** quickstart that ships with JBoss EAP.

EJB Client Code Example

```
package org.jboss.as.quickstarts.ejb.multi.server;
```

```
import java.util.Date;
import java.util.Properties;
import java.util.logging.Level;
import java.util.logging.Logger;

import javax.naming.Context;
import javax.naming.InitialContext;

import org.jboss.as.quickstarts.ejb.multi.server.app.MainApp;
import org.jboss.ejb.client.ContextSelector;
import org.jboss.ejb.client.EJBClientConfiguration;
import org.jboss.ejb.client.EJBClientContext;
import org.jboss.ejb.client.PropertiesBasedEJBClientConfiguration;
import org.jboss.ejb.client.remoting.ConfigBasedEJBClientContextSelector;

public class Client {

/**
* @param args no args needed
* @throws Exception
*/
    public static void main(String[] args) throws Exception {
        // suppress output of client messages
        Logger.getLogger("org.jboss").setLevel(Level.OFF);
        Logger.getLogger("org.xnio").setLevel(Level.OFF);

        Properties p = new Properties();

p.put("remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABL
ED", "false");
        p.put("remote.connections", "one");
        p.put("remote.connection.one.port", "8080");
        p.put("remote.connection.one.host", "localhost");
        p.put("remote.connection.one.username", "quickuser");
        p.put("remote.connection.one.password", "quick-123");

        EJBClientConfiguration cc = new
PropertiesBasedEJBClientConfiguration(p);
        ContextSelector<EJBClientContext> selector = new
ConfigBasedEJBClientContextSelector(cc);
        EJBClientContext.setSelector(selector);

        Properties props = new Properties();
        props.put(Context.URL_PKG_PREFIXES,
"org.jboss.ejb.client.naming");
        InitialContext context = new InitialContext(props);


        final String rcal = "ejb:jboss-ejb-multi-server-app-main/ejb//" +
("MainAppBean") + "!" + MainApp.class.getName();
        final MainApp remote = (MainApp) context.lookup(rcal);
        final String result = remote.invokeAll("Client call at "+new
Date());

        System.out.println("InvokeAll succeed: "+result);
```

```
    }

}
```

NOTE

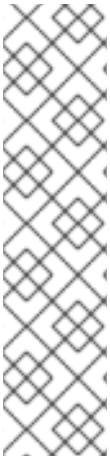
Obtaining a **`UserTransaction`** reference on the client is unsupported for scenarios with a scoped EJB client context and for invocations which use the **`remote-naming`** protocol. This is because in these scenarios, **`InitialContext`** encapsulates its own EJB client context instance; which cannot be accessed using the static methods of the EJBClient class. When EJBClient.getUserTransaction() is called, it returns a transaction from default (global) EJB client context (which might not be initialized) and not from the desired one.

UserTransaction Reference on the Client-side

The following example shows how to get **`UserTransaction`** reference on a standalone client.

UserTransaction Code Example

```
import org.jboss.ejb.client.EJBClient;
import javax.transaction.UserTransaction;
...
    Context context = null;
    UserTransaction tx = null;
    try {
      Properties props = new Properties();
      // REMEMBER: there must be a jboss-ejb-client.properties with the
      // connection parameter in the client's classpath
      props.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
      context = new InitialContext(props);
      System.out.println("\n\tGot initial Context: "+context);
      tx = EJBClient.getUserTransaction("yourServerName");
      System.out.println("UserTransaction = "+tx.getStatus());
      tx.begin();
      // do some work
      ...
    } catch (Exception e) {
      e.printStackTrace();
      tx.rollback();
    } finally{
      if(context != null) {
        context.close();
      }
    }
```



NOTE

To get **`UserTransaction`** reference on the client side; start your server with the following system property **`-Djboss.node.name=yourServerName`** and then use it on client side as following:

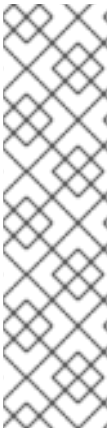
```
tx = EJBClient.getUserTransaction("yourServerName");
```

Replace "yourServerName" with the name of your server. If a user transaction is started on a node all invocations are sticky on the node and the node must have all the needed EJBs. It is not possible to use **`UserTransaction`** with remote-naming protocol and scoped-context.

Internet Inter-ORB Protocol (IIOP) Remote Call
To invoke an EJB bean using an IIOP remote call, you must first enable IIOP on the server.

To enable IIOP you must have the **`iiop-openjdk`** subsystem installed and the **`<iiop/>`** element present in the **`ejb3`** subsystem configuration. The **`standalone-full.xml`** configuration that comes with the distribution has both of these enabled.

For a bean to be reachable by IIOP remote call, it needs to use EJB 2 and home interface with narrowing. More details on IIOP remote invocation can be seen at Configure IIOP for Remote EJB Calls.



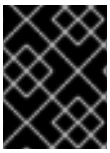
NOTE

The major differences between IIOP remote call and EJB remoting call are:

- When a client intends to invoke an EJB bean via IIOP remote call to start transaction on client side, the JTS transaction implementation has to be used. On the other hand, if a client intends to invoke an EJB bean via EJB remoting call, the JTA transaction implementation has to be used.
- For EJB invocation via IIOP remote call, the transaction is created on client and propagated via call to the server. Whereas, for EJB invocation via EJB remoting call, the transaction is looked up on the server and is managed on the client side.

To enable JTS transactions on the server, you must change the **`transactions`** attribute from value **`spec`** to **`full`** in the **`iiop-openjdk`** subsystem and set the **`jts`** attribute in the **`transactions`** subsystem to **`true`**. You can accomplish this by using the following management CLI commands.

```
/subsystem=iiop-openjdk/:write-attribute(name=transactions,value=full)
/subsystem=transactions/:write-attribute(name=jts,value=true)
```



IMPORTANT

For the client to successfully invoke an EJB transaction using an IIOP call, we need to add the client-side dependency on **`org.wildfly:wildfly-iiop-openjdk`**.

IIOP Client Code Example

```
import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
```

```
import javax.rmi.PortableRemoteObject;
import com.arjuna.ats.arjuna.recovery.RecoveryManager;
import com.arjuna.ats.internal.jts.ORBManager;
import com.arjuna.ats.internal.jts.context.ContextPropagationManager;
import com.arjuna.ats.jts.OTSManager;
import com.sun.corba.se.impl.orbutil.ORBConstants;
import com.arjuna.orbportability.ORB;
import com.arjuna.orbportability.OA;

final String host = "localhost";
final int port = 3528;

// For client we define how the Narayana will behave
System.setProperty("com.arjuna.ats.jts.alwaysPropagateContext", "true");

// Set orb to be initialized on client and being able to start ORB txn
Properties properties = new Properties();
properties.setProperty(ORBConstants.PERSISTENT_SERVER_PORT_PROPERTY,
"15151");
properties.setProperty(ORBConstants.ORB_SERVER_ID_PROPERTY, "1");

// registers the appropriate filter with the ORB
new ContextPropagationManager();

org.omg.CORBA.ORB sunOrb = org.omg.CORBA.ORB.init(new String[0],
properties);
ORB orb = null;
try {
    orb = com.arjuna.orbportability.ORB.getInstance("ClientSide");
    orb.setOrb(sunOrb);

    OA oa = OA.getRootOA(orb);
    org.omg.PortableServer.POA rootPOA =
org.omg.PortableServer.POAHelper.narrow(sunOrb.resolve_initial_references(
"RootPOA"));
    oa.setPOA(rootPOA);

    oa.initOA();

    ORBManager.setORB(orb);
    ORBManager.setPOA(oa);

    // Recovery manager has to be started on client when we want recovery
to work at client
    RecoveryManager.manager().startRecoveryManagerThread();

    // Getting context to lookup
    System.setProperty("com.sun.CORBA.ORBUseDynamicStub", "true");
    final Properties prope = new Properties();
    prope.put(Context.PROVIDER_URL, "corbaloc::" + host + ":" + port +
"/JBoss/Naming/root");
    prope.setProperty(Context.URL_PKG_PREFIXES,
"org.jboss.iiop.naming:org.jboss.naming.client");
    prope.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.cosnaming.CNCtxFactory");
    Context context = new InitialContext(prope);
```

```
    // Bean lookup
    final Object iiopObj =
context.lookup(IIOPBeanMandatory.class.getSimpleName());
    final IIOPBeanHome beanHome = (IIOPBeanHome)
PortableRemoteObject.narrow(iiopObj, IIOPBeanHome.class);
    final IIOPRemote bean = beanHome.create();

    // Starting orb transaction
    OTSManager.get_current().begin();

    // Call bean - business logic
    bean.sayHello();

    // Manage the commit of the work
    OTSManager.get_current().commit(true);
    // or rollback
    // OTSManager.get_current().rollback();
} finally {
    // It's good to release resources - do it only once at the end
    if (orb != null) {
        orb.shutdown();
    }
    RecoveryManager.manager().terminate();
}
```

For more information, see Configuring Transactions in the JBoss EAP *Configuration Guide*.

5.5. EXAMPLE EJB INVOCATION FROM A REMOTE SERVER INSTANCE

JBoss EAP is secure by default. No communication can happen with a server instance from a remote client without passing the appropriate credentials, irrespective of whether it is a standalone client or another server instance. In order to allow a client server to communicate with a destination server, you must configure user credentials to be used during this server communication.

Following example demonstrates how to invoke EJBs deployed on a JBoss EAP server instance from another remote JBoss EAP server instance. For ease of reference, let us use the following aliases:

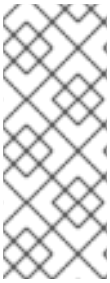
- Client server: the server from which the EJB invocation happens.
- Destination server: the server on which the EJB is deployed.

Prerequisites

- Configure the user with required credentials on the destination server. See Adding a Management User in the JBoss EAP *Configuration Guide* for details.
- Start the destination server.

  ```
  ./standalone.sh -server-config=standalone-full.xml
  ```

- Deploy the application. See Deploying Applications in the JBoss EAP *Configuration Guide* for details.



NOTE

Each of your server instances must have a unique **jboss.node.name** system property. You can set this value by passing it to the startup script:

```
./standalone.sh -server-config=standalone-full.xml -
Djboss.node.name=<add appropriate value here>
```

5.5.1. Configuring the Client Server

You must let the client server know about the destination server's EJB remoting connector, over which it can communicate during the EJB invocations. To achieve this, you must add a **remote-outbound-connection** to the **remoting** subsystem on the client server. The **remote-outbound-connection** configuration indicates that an outbound connection will be created to a remote server instance from this client server. The **remote-outbound-connection** must have an **outbound-socket-binding** configured with itself, which points to a remote host and a remote port of the destination server.

1. Start the client server:

```
/standalone.sh -server-config=standalone-full.xml -
Djboss.socket.binding.port-offset=100
```

2. Create a security realm on the client server to communicate with a secure destination server. The client server must provide the user credentials to the destination server. To achieve this, you need to create a security realm on the client server, which will pass the user information provided for the user that was added to the destination server.
   You must use a security realm which stores a base64-encoded password and then passes on these credentials when asked for. You need to create the base64 encoded version of the password that was provided for the user created initially for the destination server. You may use OpenSSL to generate base64-encoded passwords at the command line.

```
echo -n “password” | openssl dgst -sha256 -binary | openssl base64
```

   Here the password in plain text - **password** - is piped into the OpenSSL digest function then piped into another OpenSSL function to convert into base64-encoded format. You can now use base64-encoded password in the security realm that you configure on the client server.

3. Run the following management CLI commands to create a security realm for the base64-encoded password:

```
/core-service=management/security-realm=ejb-security-realm:add()
/core-service=management/security-realm=ejb-security-realm/server-
identity=secret:add(value=<base64-encoded password>)
```

   You may notice that the management CLI shows the message **"process-state" ⇒ "reload-required"**, so you must restart the server before this change can be used.

   On successful invocation of this command, the following configuration will be created in the **<management>** section of **standalone.xml**:

```
<management>
        <security-realms>
            ...
            <security-realm name="ejb-security-realm">
```

```
                <server-identities>
                    <secret value=<base64-encoded password>/>
                </server-identities>
            </security-realm>
        </security-realms>
...
```

The code snippet above creates a security realm named **ejb-security-realm** with the base64-encoded password.

4. Create an outbound-socket-binding on the client server. You must now create an **outbound-socket-binding** that points to the destination server's host and port.

```
/socket-binding-group=standard-sockets/remote-destination-outbound-
socket-binding=remote-ejb:add(host=localhost, port=8080)
```

The command above creates an **outbound-socket-binding** named **remote-ejb** which points to **localhost** as the host and port 8080 as the destination port. Note that the host information should match the host/IP of the destination server. In this example, we are running the client and destination servers on the same machine so we use **localhost**. Similarly, the port information should match the **http-remoting** connector port used by the **ejb3** subsystem; by default it is 8080.

When this command is run successfully, you will see that the **standalone-full.xml** was updated with the following **outbound-socket-binding** in the **socket-binding-group**:

```
<socket-binding-group name="standard-sockets" default-
interface="public" port-offset="${jboss.socket.binding.port-
offset:0}">
       ...
       <outbound-socket-binding name="remote-ejb">
          <remote-destination host="localhost" port="8080"/>
       </outbound-socket-binding>
   </socket-binding-group>
```

5. Create a **remote-outbound-connection** that uses this newly created **outbound-socket-binding**. Now let us create a **remote-outbound-connection** which will use the newly created **outbound-socket-binding** pointing to the EJB remoting connector of the destination server:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-
connection:add(outbound-socket-binding-ref=remote-ejb,
protocol=http-remoting, security-realm=ejb-security-realm,
username=ejb)
```

The command above creates a **remote-outbound-connection** named **remote-ejb-connection** in the **remoting** subsystem and uses the previously created **remote-ejb outbound-socket-binding**. Notice the **outbound-socket-binding-ref** in the command above, with the **http-remoting** protocol. Furthermore, we also set the **security-realm** attribute to point to the security realm that we created in the previous step. Also notice that the **username** attribute is set to use the user who is allowed to communicate with the destination server.

This step creates an outbound connection on the client server to the remote destination server

and sets up the username to the user who is allowed to communicate with that destination server. It also sets up the security realm to a pre-configured security realm capable of passing along the user credentials (in this case the password). This way when a connection has to be established from the client server to the destination server, the connection creation logic will have the necessary security credentials to pass along and set up a successful secure connection.

Let us run the following two operations to set some default connection creation options for the outbound connection:

```
/subsystem=remoting/remote-outbound-connection=remote-ejb-
connection/property=SASL_POLICY_NOANONYMOUS:add(value=false)
/subsystem=remoting/remote-outbound-connection=remote-ejb-
connection/property=SSL_ENABLED:add(value=false)
```

Ultimately, upon successful invocation of this command, the following configuration will be created in the **remoting** subsystem:

```
<subsystem xmlns="urn:jboss:domain:remoting:1.1">
...
   <outbound-connections>
        <remote-outbound-connection name="remote-ejb-connection"
outbound-socket-binding-ref="remote-ejb" protocol="http-remoting"
security-realm="ejb-security-realm" username="ejb">
            <properties>
                <property name="SASL_POLICY_NOANONYMOUS"
value="false"/>
                <property name="SSL_ENABLED" value="false"/>
            </properties>
        </remote-outbound-connection>
    </outbound-connections>
</subsystem>
```

This completes our configuration on the client server. Our next step is to deploy the application on the client server which will invoke the bean deployed on the destination server.

5.5.2. Adding jboss-ejb-client.xml to Client Application

Add the **jboss-ejb-client.xml** to the client application as **META-INF/jboss-ejb-client.xml**:

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.0">
   <client-context>
       <ejb-receivers>
           <remoting-ejb-receiver outbound-connection-ref="remote-ejb-
connection"/>
       </ejb-receivers>
   </client-context>
</jboss-ejb-client>
```

Notice that we have configured the EJB client context for this application to use a **remoting-ejb-receiver** that points to the **remote-outbound-connection** named **remote-ejb-connection**, which we had created earlier. This links the EJB client context to use the **remote-ejb-connection** pointing to the EJB remoting connector on the destination server.

5.5.3. Invoking the Bean

Following snippet shows how to invoke the bean:

```
import javax.naming.Context;
import java.util.Hashtable;
import javax.naming.InitialContext;
…
public void invokeOnBean() {
        try {
            final Hashtable props = new Hashtable();
            // setup the ejb: namespace URL factory
            props.put(Context.URL_PKG_PREFIXES,
"org.jboss.ejb.client.naming");
            // create the InitialContext
            final Context context = new
javax.naming.InitialContext(props);
            // Lookup the Greeter bean using the ejb: namespace syntax
which is explained here
https://docs.jboss.org/author/display/AS71/EJB+invocations+from+a+remote+c
lient+using+JNDI
            final Greeter bean = (Greeter) context.lookup("ejb:" +
"myapp" + "/" + "myejb" + "/" + "" + "/" + "GreeterBean" + "!" +
org.myapp.ejb.Greeter.class.getName());
            // invoke on the bean
            final String greeting = bean.greet("Tom");
            System.out.println("Received greeting: " + greeting);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
}
```

The code above will invoke the bean deployed on the destination server and return the result.

5.5.4. Deploying the Client Application

Let us deploy the client application on the client server. You can use either the CLI or the admin console or a IDE or deploy manually to **`EAP_HOME/standalone/deployments`** folder. Ensure that the application is deployed successfully.

See Deploying Applications in the JBoss EAP *Configuration Guide* for details.

5.6. USING SCOPED EJB CLIENT CONTEXTS

Summary

To invoke an EJB In earlier versions of JBoss EAP, you would typically create a JNDI context and pass it the PROVIDER_URL, which would point to the target server. Any invocations done on EJB proxies that were looked up using that JNDI context, would end up on that server. With scoped EJB client contexts, user applications have control over which EJB receiver is used for a specific invocation.

Use Scoped EJB Client Context in a Remote Standalone Client

Prior to the introduction of scoped EJB client contexts, the context was typically scoped to the client application. Scoped client contexts now allow the EJB client contexts to be scoped with the JNDI contexts. The following is an example of a standalone remote client application that invokes the same

bean twice using a scoped EJB client context:

```
public class MyApplication {
    public static void main(String args[]) {

        // Use the "foo" security credential connect to the server and
invoke this bean instance
        final Properties ejbClientContextPropsOne =
getPropsForEJBClientContextOne():
        final javax.naming.Context ctxOne = new
javax.naming.InitialContext(ejbClientContextPropsOne);
        final MyBeanInterface beanOne =
ctxOne.lookup("ejb:app/module/distinct/bean!interface");
        beanOne.doSomething();
        ...
        ctxOne.close();

        // Use the "bar" security credential to connect to the server and
invoke this bean instance
        final Properties ejbClientContextPropsTwo =
getPropsForEJBClientContextTwo():
        final javax.naming.Context ctxTwo = new
javax.naming.InitialContext(ejbClientContextPropsTwo);
        final MyBeanInterface beanTwo =
ctxTwo.lookup("ejb:app/module/distinct/bean!interface");
        beanTwo.doSomething();
        ...
        ctxTwo.close();
    }
}
```

To use the scoped EJB client context, you configure EJB client properties programmatically and pass the properties on context creation. The properties are the same set of properties that are used in the standard **`jboss-ejb-client.properties`** file. To scope the EJB client context to the JNDI context, you must also specify the **`org.jboss.ejb.client.scoped.context`** property and set its value to **`true`**. This property notifies the EJB client API that it must create an EJB client context, which is backed by EJB receivers, and that the created context is then scoped or visible only to the JNDI context that created it. Any EJB proxies looked up or invoked using this JNDI context will only know of the EJB client context associated with this JNDI context. Other JNDI contexts used by the application to lookup and invoke EJBs will not know about the other scoped EJB client contexts.

JNDI contexts that do not pass the **`org.jboss.ejb.client.scoped.context`** property and are not scoped to an EJB client context will use the default behavior, which is to use the existing EJB client context that is typically tied to the entire application.

Scoped EJB client contexts provide user applications with the flexibility that was associated with the JNP based JNDI invocations in previous versions of JBoss EAP. It provides user applications with more control over which JNDI context communicates to which server and how it connects to that server.

NOTE

With the scoped context, the underlying resources are no longer handled by the container or the API, so you must close the **`InitialContext`** when it is no longer needed. When the **`InitialContext`** is closed, the resources are released immediately. The proxies that are bound to it are no longer valid and any invocation will throw an Exception. Failure to close the **`InitialContext`** may result in resource and performance issues.

5.6.1. Configure EJBs Using a Scoped EJB Client Context

EJBs can be configured using a map-based scoped context. This is achieved by programmatically populating a properties map using the standard properties found in **`jboss-ejb-client.properties`**, specifying **`true`** for the **`org.jboss.ejb.client.scoped.context`** property, and passing the properties on the **`InitialContext`** creation.

The benefit of using a scoped context is that it allows you to configure access without directly referencing the EJB or importing JBoss classes. It also provides a way to configure and load balance a host at runtime in a multithreaded environment.

Configure an EJB Using a Map-Based Scoped Context

1. Set the properties:
   Configure the EJB client properties programmatically, specifying the same set of properties that are used in the standard **`jboss-ejb-client.properties`** file. To enable the scoped context, you must specify the **`org.jboss.ejb.client.scoped.context`** property and set its value to **`true`**. Following is an example that configures the properties programmatically.

   ```
   // Configure  EJB Client properties for the InitialContext
   Properties ejbClientContextProps = new Properties();
   ejbClientContextProps.put("remote.connections","name1");
   ejbClientContextProps.put("remote.connection.name1.host","localhost"
   );
   ejbClientContextProps.put("remote.connection.name1.port","8080");
   // Property to enable scoped EJB client context which will be tied
   to the JNDI context
   ejbClientContextProps.put("org.jboss.ejb.client.scoped.context",
   "true");
   ```

2. Pass the properties on the context creation:

   ```
   // Create the context using the configured properties
   InitialContext ic = new InitialContext(ejbClientContextProps);
   MySLSB bean = ic.lookup("ejb:myapp/ejb//MySLSBBean!" +
   MySLSB.class.getName());
   ```

3. Close the scoped EJB client context:
   Look up the root JNDI context for **`ejb:`** string to fetch the EJB naming context. Then use **`ejbRootNamingContext`** instance to look up the rest of the EJB JNDI name to fetch the EJB proxy. Use the **`close()`** method to close **`ejbRootNamingContext`** and the EJB JNDI context. Closing **`ejbRootNamingContext`** ensures that the scoped EJB client context associated with the JNDI context is closed too. Effectively, this closes the connection to the server within that EJB client context.

   ```
   final Properties props = new Properties();
   ```

```
// mark it for scoped EJB client context
props.put("org.jboss.ejb.client.scoped.context","true");
// add other properties
props.put(...);
...
Context jndiCtx = new InitialContext(props);
Context ejbRootNamingContext = (Context) jndiCtx.lookup("ejb:");
try {
    final MyBean bean =
(MyBean)ejbRootNamingContext.lookup("app/module/distinct/bean!interf
ace");
} finally {
    try { // close the EJB naming JNDI context
        ejbRootNamingContext.close();
    } catch (Throwable t) {
        // log and ignore
    }
    try { // also close our other JNDI context since we are done
with it too
        jndiCtx.close();
    } catch (Throwable t) {
        // log and ignore
    }
}
```

Contexts generated by lookup EJB proxies are bound by this scoped context and use only the relevant connection parameters. This makes it possible to create different contexts to access data within a client application or to independently access servers using different logins.

In the client, both the scoped **`InitialContext`** and the scoped proxy are passed to threads, allowing each thread to work with the given context. It is also possible to pass the proxy to multiple threads that can use it concurrently.

The scoped context EJB proxy is serialized on the remote call and then deserialized on the server. When it is deserialized, the scoped context information is removed and it returns to its default state. If the deserialized proxy is used on the remote server, because it no longer has the scoped context that was used when it was created, this can result in an **`EJBCLIENT000025`** error or possibly call an unwanted target by using the EJB name.

5.7. EJB CLIENT PROPERTIES

Summary

The following tables list properties that can be configured programmatically or in the **`jboss-ejb-client.properties`** file.

EJB Client Global Properties

The following table lists properties that are valid for the whole library within the same scope.

Table 5.1. Global Properties

Property Name	Description

Property Name	Description
`endpoint.name`	Name of the client endpoint. If not set, the default value is **`client-endpoint`**. This can be helpful to distinguish different endpoint settings because the thread name contains this property.
`remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED`	Boolean value that specifies whether the SSL protocol is enabled for all connections.  **WARNING** Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.
`deployment.node.selector`	The fully qualified name of the implementation of **`org.jboss.ejb.client.DeploymentNodeSelector`**. This is used to load balance the invocation for the EJBs.
`invocation.timeout`	The timeout for the EJB handshake or method invocation request/response cycle. The value is in milliseconds. The invocation of any method throws a **`java.util.concurrent.TimeoutException`** if the execution takes longer than the timeout period. The execution completes and the server is not interrupted.
`reconnect.tasks.timeout`	The timeout for the background reconnect tasks. The value is in milliseconds. If a number of connections are down, the next client EJB invocation will use an algorithm to decide if a reconnect is necessary to find the right node.

Property Name	Description
`org.jboss.ejb.client.scoped.context`	Boolean value that specifies whether to enable the scoped EJB client context. The default value is false. If set to **`true`**, the EJB Client will use the scoped context that is tied to the JNDI context. Otherwise the EJB client context will use the global selector in the JVM to determine the properties used to call the remote EJB and host.

EJB Client Connection Properties

The connection properties start with the prefix **`remote.connection.`** where the CONNECTION_NAME is a local identifier only used to uniquely identify the connection.

Table 5.2. Connection Properties

Property Name	Description
`remote.connections`	A comma-separated list of active **`connection-names`**. Each connection is configured by using this name.
`remote.connection.CONNECTION_NAME.host`	The host name or IP for the connection.
`remote.connection.CONNECTION_NAME.port`	The port for the connection. The default value is **`8080`**.
`remote.connection.CONNECTION_NAME.username`	The user name used to authenticate connection security.
`remote.connection.CONNECTION_NAME.password`	The password used to authenticate the user.
`remote.connection.CONNECTION_NAME.connect.timeout`	The timeout period for the initial connection. After that, the reconnect task will periodically check whether the connection can be established. The value is in milliseconds.
`remote.connection.CONNECTION_NAME.callback.handler.class`	Fully qualified name of the CallbackHandler class. It will be used to establish the connection and can not be changed as long as the connection is open.
`remote.connection.CONNECTION_NAME.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES`	Integer value specifying the maximum number of outbound requests. The default is **`80`**. There is only one connection from the client (JVM) to the server to handle all invocations.

Property Name	Description
remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOUS	Boolean value that determines whether credentials must be provided by the client to connect successfully. The default value is **true**. If set to **true**, the client must provide credentials. If set to **false**, invocation is allowed as long as the remoting connector does not request a security realm.
remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SASL_DISALLOWED_MECHANISMS	Disables certain SASL mechanisms used for authenticating during connection creation. **JBOSS-LOCAL-USER** means the silent authentication mechanism, used when the client and server are on the same machine, is disabled.
remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SASL_POLICY_NOPLAINTEXT	**Boolean** value that enables or disables the use of plain text messages during the authentication. If using JAAS, it must be set to false to allow a plain text password.
remote.connection.CONNECTION_NAME.connect.options.org.xnio.Options.SSL_ENABLED	**Boolean** value that specifies whether the SSL protocol is enabled for this connection. **WARNING** Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.
remote.connection.CONNECTION_NAME.connect.options.org.jboss.remoting3.RemotingOptions.HEARTBEAT_INTERVAL	Interval to send a heartbeat between client and server to prevent automatic close, for example, in the case of a firewall. The value is in milliseconds.

EJB Client Cluster Properties

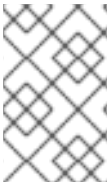
If the initial connection connects to a clustered environment, the topology of the cluster is received automatically and asynchronously. These properties are used to connect to each received member. Each property starts with the prefix **remote.cluster.** where the CLUSTER_NAME refers to the related to the servers **infinispan** subsystem configuration.

Table 5.3. Cluster Properties

Property Name	Description
remote.cluster.CLUSTER_NAME.clusternode.selector	The fully qualified name of the implementation of **org.jboss.ejb.client.ClusterNodeSelector**. This class, rather than **org.jboss.ejb.client.DeploymentNodeSelector**, is used to load balance EJB invocations in a clustered environment. If the cluster is completely down, the invocation will fail with the message *No ejb receiver available*.
remote.cluster.CLUSTER_NAME.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES	Integer value specifying the maximum number of outbound requests that can be made to the entire cluster.
remote.cluster.CLUSTER_NAME.node.NODE_NAME.channel.options.org.jboss.remoting3.RemotingOptions.MAX_OUTBOUND_MESSAGES	Integer value specifying the maximum number of outbound requests that can be made to this specific cluster-node.

5.8. REMOTE EJB DATA COMPRESSION

Previous versions of JBoss EAP included a feature where the message stream that contained the EJB protocol message could be compressed. This feature has been included in JBoss EAP 6.3 and later.



NOTE

Compression currently can only be specified by annotations on the EJB interface which should be on the client and server side. There is not currently an XML equivalent to specify compression hints.

Data compression hints can be specified via the JBoss annotation org.jboss.ejb.client.annotation.CompressionHint. The hint values specify whether to compress the request, response or request and response. Adding **@CompressionHint** defaults to **compressResponse=true** and **compressRequest=true**.

The annotation can be specified at the interface level to apply to all methods in the EJB's interface such as:

```
import org.jboss.ejb.client.annotation.CompressionHint;

@CompressionHint(compressResponse = false)
public interface ClassLevelRequestCompressionRemoteView {
    String echo(String msg);
}
```

Or the annotation can be applied to specific methods in the EJB's interface such as:

```
import org.jboss.ejb.client.annotation.CompressionHint;
```

```
public interface CompressableDataRemoteView {

    @CompressionHint(compressResponse = false, compressionLevel =
Deflater.BEST_COMPRESSION)
    String echoWithRequestCompress(String msg);

    @CompressionHint(compressRequest = false)
    String echoWithResponseCompress(String msg);

    @CompressionHint
    String echoWithRequestAndResponseCompress(String msg);

    String echoWithNoCompress(String msg);
}
```

The **`compressionLevel`** setting shown above can have the following values:

- BEST_COMPRESSION
- BEST_SPEED
- DEFAULT_COMPRESSION
- NO_COMPRESSION

The **`compressionLevel`** setting defaults to **`Deflater.DEFAULT_COMPRESSION`**.

Class level annotation with method level overrides:

```
@CompressionHint
public interface MethodOverrideDataCompressionRemoteView {

    @CompressionHint(compressRequest = false)
    String echoWithResponseCompress(final String msg);

    @CompressionHint(compressResponse = false)
    String echoWithRequestCompress(final String msg);

    String echoWithNoExplicitDataCompressionHintOnMethod(String msg);
}
```

On the client side ensure the **`org.jboss.ejb.client.view.annotation.scan.enabled`** system property is set to **`true`**. This property tells JBoss EJB Client to scan for annotations.

5.9. EJB CLIENT REMOTING INTEROPERABILITY

The default remote connection port is **`8080`**. The **`jboss-ejb-client`** properties file looks like this:

```
remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=fals
e
remote.connections=default
remote.connection.default.host=localhost
```

```
remote.connection.default.port=8080
remote.connection.default.connect.options.org.xnio.Options.SASL_POLICY_NOA
NONYMOUS=false
```

Default Connector

The default connector is **`http-remoting`**.

- If a client application uses the EJB client library from JBoss EAP 6 and wants to connect to a JBoss EAP 7 server, the server must be configured to expose a remoting connector on a port other than **`8080`**. The client must then connect using that newly configured connector.
- A client application that uses the EJB client library from JBoss EAP 7 and wants to connect to a JBoss EAP 6 server must be aware that the server instance does not use the **`http-remoting`** connector and instead uses a **`remoting`** connector. This is achieved by defining a new client-side connection property.

  ```
  remote.connection.default.protocol=remote
  ```

NOTE

EJB remote calls are supported for JBoss EAP 7 with JBoss EAP 6 only.

Besides EJB client remoting interoperability, you can connect to legacy clients using the following options:

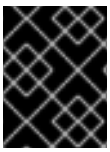
- Configure the ORB for JTS Transactions in the JBoss EAP *Configuration Guide*.

5.10. CONFIGURE IIOP FOR REMOTE EJB CALLS

JBoss EAP supports CORBA/IIOP-based access to EJBs deployed on JBoss EAP.

The **`<iiop>`** element is used to enable IIOP, CORBA, invocation of EJBs. The presence of this element means that the **`iiop-openjdk`** subsystem is installed. The **`<iiop>`** element includes the following two attributes:

- **`enable-by-default`**: If this is **`true`**, then all the EJBs with EJB 2.x home interfaces are exposed through IIOP. Otherwise they must be explicitly enabled through **`jboss-ejb3.xml`**.
- **`use-qualified-name`**: If this is **`true`**, then the EJBs are bound to the CORBA naming context with a binding name that contains the application and modules name of the deployment, such as **`myear/myejbjar/MyBean`**. If this is **`false`**, then the default binding name is simply the bean name.



IMPORTANT

IIOP calls can be done only with EJB 2 beans. EJB 3 beans are not supported by IIOP in JBoss EAP 7.0.

Enabling IIOP

To enable IIOP you must have the IIOP OpenJDK ORB subsystem installed, and the **`<iiop/>`** element present in the **`ejb3`** subsystem configuration. The **`standalone-full.xml`** configuration that comes with the distribution has both of these enabled.

IIOP is configured in the **iiop-openjdk** subsystem of the server configuration file.

```
<subsystem xmlns="urn:jboss:domain:iiop-openjdk:1.0">
```

Use the following management CLI command to access and update the **iiop-openjdk** subsystem.

```
/subsystem=iiop-openjdk
```

The IIOP element takes two attributes that control the default behavior of the server.

```
<subsystem xmlns="urn:jboss:domain:ejb3:1.2">
  ...
  <iiop enable-by-default="false" use-qualified-name="false"/>
  ...
</subsystem>
```

The following management CLI command adds the **<iiop>** element under the **ejb3** subsystem:

```
/subsystem=ejb3/service=iiop:add(enable-by-default=false, use-qualified-
name=false)
```

Create an EJB That Communicates Using IIOP

The following example demonstrates how to make a remote IIOP call from the client:

1. Create an EJB 2 bean on the server:

```
@Remote(IIOPRemote.class)
@RemoteHome(IIOPBeanHome.class)
@Stateless
public class IIOPBean {
    public String sayHello() throws RemoteException {
         return "hello";
    }
}
```

2. Create a home implementation, which has a mandatory method **create()**. This method is called by the client to obtain proxy of remote interface to invoke business methods:

```
public interface IIOPBeanHome extends EJBHome {
    public IIOPRemote create() throws RemoteException;
}
```

3. Create a remote interface for remote connection to the EJB:

```
public interface IIOPRemote extends EJBObject {
    String sayHello() throws RemoteException;
}
```

4. Introduce the bean for remote call by creating a descriptor file **jboss-ejb3.xml** in **META-INF**:

```
<?xml version="1.0" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
               xmlns="http://java.sun.com/xml/ns/javaee"
```

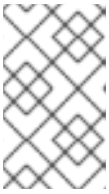
```
                xmlns:iiop="urn:iiop"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd
                   http://java.sun.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-spec-2_0.xsd
                   urn:iiop jboss-ejb-iiop_1_0.xsd"
                version="3.1"
                impl-version="2.0">
    <assembly-descriptor>
        <iiop:iiop>
            <ejb-name>*</ejb-name>
        </iiop:iiop>
    </assembly-descriptor>
</jboss:ejb-jar>
```

NOTE

The packed beans along with the descriptor in the JAR file is now ready to be deployed to the JBoss EAP container.

5. Create a context at the client side:

```
System.setProperty("com.sun.CORBA.ORBUseDynamicStub", "true");
final Properties props = new Properties();
props.put(Context.PROVIDER_URL,
"corbaloc::localhost:3528/JBoss/Naming/root");
props.setProperty(Context.URL_PKG_PREFIXES,
"org.jboss.iiop.naming:org.jboss.naming.client");
props.put(Context.INITIAL_CONTEXT_FACTORY,
"com.sun.jndi.cosnaming.CNCtxFactory");
props.put(Context.OBJECT_FACTORIES,
"org.jboss.tm.iiop.client.IIOPClientUserTransactionObjectFactory");
```



NOTE

The client will need to have the **`wildfly iiop openjdk`** library added to its class path. The client might also need to add the **`org.wildfly:wildfly-iiop-openjdk`** artifact as Maven dependency.

6. Use the context lookup to narrow the reference to the **`IIOPBeanHome`** home interface. Then call the home interface **`create()`** method to access the remote interface, which allows you to call its methods:

```
try {
    Context context = new InitialContext(props);

    final Object iiopObj =
context.lookup(IIOPBean.class.getSimpleName());
    final IIOPBeanHome beanHome = (IIOPBeanHome)
PortableRemoteObject.narrow(iiopObj, IIOPBeanHome.class);
```

```
    final IIOPRemote bean = beanHome.create();

    System.out.println("Bean saying: " + bean.sayHello());
} catch (Exception e) {
    e.printStackTrace();
}
```

CHAPTER 6. EJB APPLICATION SECURITY

6.1. SECURITY IDENTITY

6.1.1. About EJB Security Identity

An EJB can specify an identity to use when invoking methods on other components. This is the EJB security identity, also known as invocation identity.

By default, the EJB uses its own caller identity. The identity can alternatively be set to a specific security role. Using specific security roles is useful when you want to construct a segmented security model - for example, restricting access to a set of components to internal EJBs only.

6.1.2. Set the Security Identity of an EJB

The security identity of the EJB is specified through the **<security-identity>** tag in the security configuration. If no **<security-identity>** tag is present, the caller identity of the EJB is used by default.

Example: Set the security identity of an EJB to be the same as its caller

This example sets the security identity for method invocations made by an EJB to be the same as the current caller's identity. This behavior is the default if you do not specify a <security-identity> element declaration.

```
<ejb-jar>
  <enterprise-beans>
     <session>
        <ejb-name>ASessionBean</ejb-name>
        <!-- ... -->
        <security-identity>
          <use-caller-identity/>
        </security-identity>
     </session>
     <!-- ... -->
  </enterprise-beans>
</ejb-jar>
```

Example: Set the security identity of an EJB to a specific role

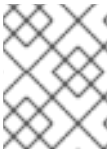
To set the security identity to a specific role, use the **<run-as>** and **<role-name>** tags inside the **<security-identity>** tag.

```
<ejb-jar>
  <enterprise-beans>
     <session>
        <ejb-name>RunAsBean</ejb-name>
        <!-- ... -->
        <security-identity>
          <run-as>
             <description>A private internal role</description>
             <role-name>InternalRole</role-name>
          </run-as>
        </security-identity>
```

```
    </session>
  </enterprise-beans>
  <!-- ... -->
</ejb-jar>
```

By default, when you use **<run-as>**, a principal named **anonymous** is assigned to outgoing calls. To assign a different principal, uses the **<run-as-principal>**.

```
<session>
    <ejb-name>RunAsBean</ejb-name>
    <security-identity>
        <run-as-principal>internal</run-as-principal>
    </security-identity>
</session>
```



NOTE

You can also use the **<run-as>** and **<run-as-principal>** elements inside a servlet element.

6.2. EJB METHOD PERMISSIONS

6.2.1. About EJB Method Permissions

EJBs can restrict access to their methods to specific security roles.

The EJB **<method-permission>** element declaration specifies the roles that can invoke the interface methods of the EJB. You can specify permissions for the following combinations:

- All home and component interface methods of the named EJB
- A specified method of the home or component interface of the named EJB
- A specified method within a set of methods with an overloaded name.

6.2.2. Use EJB Method Permissions

Overview

The **<method-permission>** element defines the logical roles that are allowed to access the EJB methods defined by **<method>** elements. Several examples demonstrate the syntax of the xml. Multiple method permission statements may be present, and they have a cumulative effect. The **<method-permission>** element is a child of the **<assembly-descriptor>** element of the **<ejb-jar>** descriptor.

The XML syntax is an alternative to using annotations for EJB method permissions.

Example: Allow roles to access all methods of an EJB

```
<method-permission>
  <description>The employee and temp-employee roles may access any method
  of the EmployeeService bean </description>
  <role-name>employee</role-name>
```

```
  <role-name>temp-employee</role-name>
  <method>
    <ejb-name>EmployeeService</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

Example: Allow roles to access only specific methods of an EJB, and limiting which method parameters can be passed

```
<method-permission>
  <description>The employee role may access the findByPrimaryKey,
  getEmployeeInfo, and the updateEmployeeInfo(String) method of
  the AcmePayroll bean </description>
  <role-name>employee</role-name>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>findByPrimaryKey</method-name>
  </method>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>getEmployeeInfo</method-name>
  </method>
  <method>
    <ejb-name>AcmePayroll</ejb-name>
    <method-name>updateEmployeeInfo</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
    </method-params>
  </method>
</method-permission>
```

Example: Allow any authenticated user to access methods of EJBs

Using the **<unchecked/>** element allows any authenticated user to use the specified methods.

```
<method-permission>
  <description>Any authenticated user may access any method of the
  EmployeeServiceHelp bean</description>
  <unchecked/>
  <method>
    <ejb-name>EmployeeServiceHelp</ejb-name>
    <method-name>*</method-name>
  </method>
</method-permission>
```

Example: Completely exclude specific EJB methods from being used

```
<exclude-list>
  <description>No fireTheCTO methods of the EmployeeFiring bean may be
  used in this deployment</description>
  <method>
    <ejb-name>EmployeeFiring</ejb-name>
```

```
    <method-name>fireTheCTO</method-name>
  </method>
</exclude-list>
```

Example: A complete `<assembly-descriptor>` containing several `<method-permission>` blocks

```
<ejb-jar>
    <assembly-descriptor>
        <method-permission>
            <description>The employee and temp-employee roles may access
any
                method of the EmployeeService bean </description>
            <role-name>employee</role-name>
            <role-name>temp-employee</role-name>
            <method>
                <ejb-name>EmployeeService</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>The employee role may access the
findByPrimaryKey,
                getEmployeeInfo, and the updateEmployeeInfo(String) method
of
                the AcmePayroll bean </description>
            <role-name>employee</role-name>
            <method>
                <ejb-name>AcmePayroll</ejb-name>
                <method-name>findByPrimaryKey</method-name>
            </method>
            <method>
                <ejb-name>AcmePayroll</ejb-name>
                <method-name>getEmployeeInfo</method-name>
            </method>
            <method>
                <ejb-name>AcmePayroll</ejb-name>
                <method-name>updateEmployeeInfo</method-name>
                <method-params>
                    <method-param>java.lang.String</method-param>
                </method-params>
            </method>
        </method-permission>
        <method-permission>
            <description>The admin role may access any method of the
                EmployeeServiceAdmin bean </description>
            <role-name>admin</role-name>
            <method>
                <ejb-name>EmployeeServiceAdmin</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <method-permission>
            <description>Any authenticated user may access any method of
the
                EmployeeServiceHelp bean</description>
```

```
            <unchecked/>
            <method>
                <ejb-name>EmployeeServiceHelp</ejb-name>
                <method-name>*</method-name>
            </method>
        </method-permission>
        <exclude-list>
            <description>No fireTheCTO methods of the EmployeeFiring bean
may be
                used in this deployment</description>
            <method>
                <ejb-name>EmployeeFiring</ejb-name>
                <method-name>fireTheCTO</method-name>
            </method>
        </exclude-list>
    </assembly-descriptor>
</ejb-jar>
```

6.3. EJB SECURITY ANNOTATIONS

6.3.1. About EJB Security Annotations

EJB **`javax.annotation.security`** annotations are defined in JSR250.

EJBs use security annotations to pass information about security to the deployer. These include:

@DeclareRoles

Declares which roles are available.

@RunAs

Configures the propagated security identity of a component.

6.3.2. Use EJB Security Annotations

Overview

You can use either XML descriptors or annotations to control which security roles are able to call methods in your Enterprise JavaBeans (EJBs). For information on using XML descriptors, refer to Use EJB Method Permissions.

Any method values explicitly specified in the deployment descriptor override annotation values. If a method value is not specified in the deployment descriptor, those values set using annotations are used. The overriding granularity is on a per-method basis.

Annotations for Controlling Security Permissions of EJBs

@DeclareRoles

Use **`@DeclareRoles`** to define which security roles to check permissions against. If no **`@DeclareRoles`** is present, the list is built automatically from the **`@RolesAllowed`** annotation. For information about configuring roles, refer to the Java EE 7 Tutorial Specifying Authorized Users by Declaring Security Roles.

@RolesAllowed, @PermitAll, @DenyAll

Use **`@RolesAllowed`** to list which roles are allowed to access a method or methods. Use

@PermitAll or **@DenyAll** to either permit or deny all roles from using a method or methods. For information about configuring annotation method permissions, refer to the Java EE 7 Tutorial Specifying Authorized Users by Declaring Security Roles.

@RunAs

Use **@RunAs** to specify a role a method uses when making calls from the annotated method. For information about configuring propagated security identities using annotations, refer to the Java EE 7 Tutorial section 49.2.3, Propagating a Security Identity (Run-As).

Example: Security Annotations Example

```
@Stateless
@RolesAllowed({"admin"})
@SecurityDomain("other")
public class WelcomeEJB implements Welcome {
    @PermitAll
    public String WelcomeEveryone(String msg) {
        return "Welcome to " + msg;
    }
    @RunAs("tempemployee")
    public String GoodBye(String msg) {
        return "Goodbye, " + msg;
    }
    public String GoodbyeAdmin(String msg) {
        return "See you later, " + msg;
    }
}
```

In this code, all roles can access method **WelcomeEveryone**. The **GoodBye** method uses the **tempemployee** role when making calls. Only the **admin** role can access method **GoodbyeAdmin**, and any other methods with no security annotation.

6.4. REMOTE ACCESS TO EJBS

6.4.1. Use Security Realms with Remote EJB Clients

One way to add security to clients which invoke EJBs remotely is to use security realms. A security realm is a simple database of username/password pairs and username/role pairs. The terminology is also used in the context of web containers, with a slightly different meaning.

To authenticate a specific username/password pair that exists in a security realm against an EJB, follow these steps:

- Add a new security realm to the domain controller or standalone server.
- Add the following parameters to the **jboss-ejb-client.properties** file, which is in the classpath of the application. This example assumes the connection is referred to as **default** by the other parameters in the file.

  ```
  remote.connection.default.username=appuser
  remote.connection.default.password=apppassword
  ```

- Create a custom Remoting connector on the domain or standalone server, which uses your new security realm.

- Deploy your EJB to the server group which is configured to use the profile with the custom Remoting connector, or to your standalone server if you are not using a managed domain.

6.4.2. Add a New Security Realm

1. Run the Management CLI:
   Start the **jboss-cli.sh** or **jboss-cli.bat** command and connect to the server.

2. Create the new security realm itself:
   Run the following command to create a new security realm named **MyDomainRealm** on a domain controller or a standalone server.

   For a domain instance, use this command:

   ```
   /host=master/core-service=management/security-
   realm=MyDomainRealm:add()
   ```

   For a standalone instance, use this command:

   ```
   /core-service=management/security-realm=MyDomainRealm:add()
   ```

3. Create a properties file named **myfile.properties**:
   For a standalone instance, create a file **EAP_HOME/standalone/configuration/myfile.properties** and for a domain instance, create a file **EAP_HOME/domain/configuration/myfile.properties**. These files need to have read and write access for the file owner.

   ```
   chmod 600 myfile.properties
   ```

4. Create the references to the properties file which will store information about the new role:
   Run the following command to create a pointer to the **myfile.properties** file, which will contain the properties pertaining to the new role.

   NOTE

   The properties file will not be created by the included **add-user.sh** and **add-user.bat** scripts. It must be created externally.

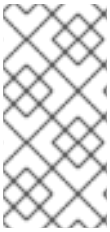
   For a domain instance, use this command:

   ```
   /host=master/core-service=management/security-
   realm=MyDomainRealm/authentication=properties:add(path=myfile.proper
   ties)
   ```

   For a standalone instance, use this command:

   ```
   /core-service=management/security-
   realm=MyDomainRealm/authentication=properties:add(path=myfile.proper
   ties)
   ```

Your new security realm is created. When you add users and roles to this new realm, the information will be stored in a separate file from the default security realms. You can manage this new file using your own applications or procedures.



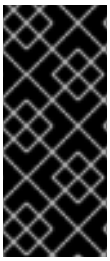
NOTE

When using the **`add-user.sh`** script to add a user to a non-default file, other than **`application-users.properties`**, you have to pass it the argument **`--user-properties myfile.properties`** otherwise it will try to use **`application-users.properties`**.

6.4.3. Add a User to a Security Realm

1. Run the **`add-user.sh`** or **`add-user.bat`** command. Open a terminal and change directories to the **`/bin/`** directory. If you are on Red Hat Enterprise Linux or any other UNIX-like operating system, run **`add-user.sh`**. If you are on Microsoft Windows Server, run **`add-user.bat`**.
2. Choose whether to add a management user or application user. For this procedure, type **`b`** to add an application user.
3. Choose the realm the user will be added to. By default, the only available realm is **`ApplicationRealm`**. If you have added a custom realm, you may add the user to that instead.
4. Type the username, password, and roles, when prompted. Type the desired username, password, and optional roles when prompted. Verify your choice by typing **`yes`**, or type **`no`** to cancel the changes. The changes are written to each of the properties files for the security realm.

6.4.4. Relationship Between Security Domains and Security Realms



IMPORTANT

For EJBs to be secured by security realms, they have to use a security domain which is configured to retrieve user credentials from the security realm. This means that the domain needs to contain the Remoting and RealmDirect login modules. Assigning a security domain is done by the **`@SecurityDomain`** annotation, which can be applied on an EJB.

The **`other`** security domain retrieves the user and password data from the underlying security realm. This security domain is the default one if there is no **`@SecurityDomain`** annotation on the EJB but the EJB contains any of the other security-related annotations to be considered secured.

The underlying **`http-remoting connector`**, which is used by the client to establish a connection, decides which security realm is used. For more information on **`http-remoting connector`**, see About the Remoting Subsystem in the JBoss EAP *Configuration Guide*.

The security realm of the default connector can be changed this way:

```
/subsystem=remoting/http-connector=http-remoting-connector:write-
attribute(name=security-realm,value=MyDomainRealm)
```

6.4.5. About Remote EJB Access Using SSL Encryption

By default, the network traffic for Remote Method Invocation (RMI) of EJB2 and EJB3 Beans is not encrypted. In instances where encryption is required, Secure Sockets Layer (SSL) can be utilized so that the connection between the client and server is encrypted. Using SSL also has the added benefit of allowing the network traffic to traverse some firewalls, depending on the firewall configuration.

WARNING

Red Hat recommends that SSLv2, SSLv3, and TLSv1.0 be explicitly disabled in favor of TLSv1.1 or TLSv1.2 in all affected packages.

CHAPTER 7. CONTAINER AND CLIENT INTERCEPTORS

7.1. ABOUT CONTAINER INTERCEPTORS

Standard Java EE interceptors, as defined by the JSR 345, Enterprise JavaBeans 3.2 specification, are expected to run after the container has completed security context propagation, transaction management, and other container provided invocation processing. This is a problem if the application must intercept a call before a specific container interceptor is run.

Positioning of the Container Interceptor in the Interceptor Chain
The container interceptors configured for an EJB are guaranteed to be run before the JBoss EAP provided security interceptors, transaction management interceptors, and other server provided interceptors. This allows specific application container interceptors to process or configure relevant context data before the invocation proceeds.

Differences Between the Container Interceptor and the Java EE Interceptor API
Although container interceptors are modeled to be similar to Java EE interceptors, there are some differences in the semantics of the API. For example, it is illegal for container interceptors to invoke the **`javax.interceptor.InvocationContext.getTarget()`** method because these interceptors are invoked long before the EJB components are set up or instantiated.

7.2. CREATE A CONTAINER INTERCEPTOR CLASS

Container interceptor classes are simple Plain Old Java Objects (POJOs). They use the **`@javax.annotation.AroundInvoke`** to mark the method that is invoked during the invocation on the bean.

The following is an example of a container interceptor class that marks the **`iAmAround`** method for invocation:

Container Interceptor Code Example

```
public class ClassLevelContainerInterceptor {
    @AroundInvoke
    private Object iAmAround(final InvocationContext invocationContext)
throws Exception {
        return this.getClass().getName() + " " +
invocationContext.proceed();
    }
}
----
```

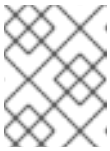
For an example of how to configure a **`jboss-ejb3.xml`** descriptor file to use a container interceptor class, see Configure a Container Interceptor.

7.3. CONFIGURE A CONTAINER INTERCEPTOR

Container interceptors use the standard Java EE interceptor libraries, meaning they use the same XSD elements that are allowed in **`ejb-jar.xml`** file for the 3.2 version of the ejb-jar deployment descriptor. Because they are based on the standard Java EE interceptor libraries, container interceptors may only be configured using deployment descriptors. This was done by design so applications would not require any JBoss specific annotation or other library dependencies. For more information about container interceptors, see About Container Interceptors.

The following procedure describes how to configure a container interceptor.

1. Create a **jboss-ejb3.xml** file in the **META-INF** directory of the EJB deployment.
2. Configure the container interceptor elements in the descriptor file.
   a. Use the **urn:container-interceptors:1.0** namespace to specify configuration of container interceptor elements.
   b. Use the **<container-interceptors>** element to specify the container interceptors.
   c. Use the **<interceptor-binding>** elements to bind the container interceptor to the EJBs. The interceptors can be bound in any of the following ways:
      - Bind the interceptor to all the EJBs in the deployment using the ***** wildcard.
      - Bind the interceptor at the individual bean level using the specific EJB name.
      - Bind the interceptor at the specific method level for the EJBs.



NOTE

These elements are configured using the EJB 3.2 XSD in the same way it is done for Java EE interceptors.

3. Review the following descriptor file for examples of the above elements.

Container Interceptor `jboss-ejb3.xml` File Example

```
<jboss xmlns="http://www.jboss.com/xml/ns/javaee"
       xmlns:jee="http://java.sun.com/xml/ns/javaee"
       xmlns:ci ="urn:container-interceptors:1.0">

    <jee:assembly-descriptor>
        <ci:container-interceptors>
            <!-- Default interceptor -->
            <jee:interceptor-binding>
                <ejb-name>*</ejb-name>
                <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.Contai
nerInterceptorOne</interceptor-class>
            </jee:interceptor-binding>
            <!-- Class level container-interceptor -->
            <jee:interceptor-binding>
                <ejb-name>AnotherFlowTrackingBean</ejb-name>
                <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ClassL
evelContainerInterceptor</interceptor-class>
            </jee:interceptor-binding>
            <!-- Method specific container-interceptor -->
            <jee:interceptor-binding>
                <ejb-name>AnotherFlowTrackingBean</ejb-name>
                <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.Method
SpecificContainerInterceptor</interceptor-class>
                <method>
```

```
                    <method-
name>echoWithMethodSpecificContainerInterceptor</method-name>
                </method>
            </jee:interceptor-binding>
            <!-- container interceptors in a specific order -->
            <jee:interceptor-binding>
                <ejb-name>AnotherFlowTrackingBean</ejb-name>
                <interceptor-order>
                    <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.ClassL
evelContainerInterceptor</interceptor-class>
                    <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.Method
SpecificContainerInterceptor</interceptor-class>
                    <interceptor-
class>org.jboss.as.test.integration.ejb.container.interceptor.Contai
nerInterceptorOne</interceptor-class>
                </interceptor-order>
                <method>
                    <method-
name>echoInSpecificOrderOfContainerInterceptors</method-name>
                </method>
            </jee:interceptor-binding>
        </ci:container-interceptors>
    </jee:assembly-descriptor>
</jboss>
```

The schema for the **`urn:container-interceptors:1.0`** namespace is available at http://www.jboss.org/schema/jbossas/jboss-ejb-container-interceptors_1_0.xsd.

7.4. CHANGE THE SECURITY CONTEXT IDENTITY

By default, when you make a remote call to an EJB that is deployed to the application server, the connection to the server is authenticated and any subsequent requests that use the connection are executed using the original authenticated identity. This is true for both client-to-server and server-to-server calls. If you need to use different identities from the same client, normally you must open multiple connections to the server so that each one is authenticated as a different identity. Rather than open multiple client connections, you can give permission to the authenticated user to switch identities and execute a request on the existing connection as a different user.

Interceptors created and configured on the server-side are referred to as container interceptors. Interceptors created and configured on the client-side are referred to as client interceptors. To change the identity of a secured connection, you must create and configure the following three components.

- Client Interceptor
- Container Interceptor
- JAAS LoginModule

The abridged code examples that follow are taken from the **`ejb-security-interceptors`** quickstart that ships with JBoss EAP. This quickstart is a simple Maven project that provides a working example of how to switch identities on an existing connection.

Create and Configure the Client Interceptor

1. Create the client interceptor.

1. Create the client interceptor.
   The client interceptor must implement the **`org.jboss.ejb.client.EJBClientInterceptor`** interface. The interceptor must pass the requested identity through the context data map, which can be obtained by using a call to **`EJBClientInvocationContext.getContextData()`**. The following is an example of a client interceptor that switches identities.

   Client Interceptor Code Example

```
public class ClientSecurityInterceptor implements
EJBClientInterceptor {

    public void handleInvocation(EJBClientInvocationContext context)
throws Exception {
        Principal currentPrincipal =
SecurityActions.securityContextGetPrincipal();

        if (currentPrincipal != null) {
            Map<String, Object> contextData =
context.getContextData();

contextData.put(ServerSecurityInterceptor.DELEGATED_USER_KEY,
currentPrincipal.getName());
        }
        context.sendRequest();
    }

    public Object handleInvocationResult(EJBClientInvocationContext
context) throws Exception {
        return context.getResult();
    }
}
```

2. Configure the client interceptor.
   An application can insert a client interceptor into the **`EJBClientContext`** interceptor chain programmatically or by using the service loader mechanism. For instructions to configure a client interceptor, see Use a Client Interceptor in an Application.

Create and Configure the Container Interceptor

Container interceptor classes are simple Plain Old Java Objects (POJOs). They use the **`@javax.annotation.AroundInvoke`** to mark the method that should be invoked during the invocation on the bean. For more information about container interceptors, see About Container Interceptors.

1. Create the container interceptor.
   This interceptor receives the **`InvocationContext`** containing the identity and makes the request to switch to that new identity. The following is an abridged version of the actual code example:

   Container Interceptor Code Example

```
public class ServerSecurityInterceptor {

    private static final Logger logger =
Logger.getLogger(ServerSecurityInterceptor.class);
```

```
    static final String DELEGATED_USER_KEY =
ServerSecurityInterceptor.class.getName() + ".DelegationUser";

    @AroundInvoke
    public Object aroundInvoke(final InvocationContext
invocationContext) throws Exception {
        Principal desiredUser = null;
        UserPrincipal connectionUser = null;

        Map<String, Object> contextData =
invocationContext.getContextData();
        if (contextData.containsKey(DELEGATED_USER_KEY)) {
            desiredUser = new SimplePrincipal((String)
contextData.get(DELEGATED_USER_KEY));

            Collection<Principal> connectionPrincipals =
SecurityActions.getConnectionPrincipals();

            if (connectionPrincipals != null) {
                for (Principal current : connectionPrincipals) {
                    if (current instanceof UserPrincipal) {
                        connectionUser = (UserPrincipal) current;
                        break;
                    }
                }

            } else {
                throw new IllegalStateException("Delegation user
requested but no user on connection found.");
            }
        }


        ContextStateCache stateCache = null;
        try {
            if (desiredUser != null && connectionUser != null
                &&
(desiredUser.getName().equals(connectionUser.getName()) == false)) {
                // The final part of this check is to verify that
the change does actually indicate a change in user.
                try {
                    // We have been requested to use an
authentication token
                    // so now we attempt the switch.
                    stateCache =
SecurityActions.pushIdentity(desiredUser, new
OuterUserCredential(connectionUser));
                } catch (Exception e) {
                    logger.error("Failed to switch security context
for user", e);
                    // Don't propagate the exception stacktrace
back to the client for security reasons
                    throw new EJBAccessException("Unable to attempt
switching of user.");
                }
```

```
            }

            return invocationContext.proceed();
        } finally {
            // switch back to original context
            if (stateCache != null) {
                SecurityActions.popIdentity(stateCache);;
            }
        }
    }
```

2. Configure the container interceptor.
   For information on how to configure container interceptors, see Configure a Container Interceptor.

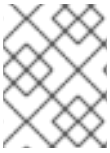
Create the JAAS LoginModule

The JAAS LoginModule component is responsible for verifying that the user is allowed to execute requests as the requested identity. The following abridged code example shows the methods that perform the login and validation:

LoginModule Code Example

```
    @SuppressWarnings("unchecked")
    @Override
    public boolean login() throws LoginException {
        if (super.login() == true) {
            log.debug("super.login()==true");
            return true;
        }

        // Time to see if this is a delegation request.
        NameCallback ncb = new NameCallback("Username:");
        ObjectCallback ocb = new ObjectCallback("Password:");

        try {
            callbackHandler.handle(new Callback[] { ncb, ocb });
        } catch (Exception e) {
            if (e instanceof RuntimeException) {
                throw (RuntimeException) e;
            }
            // If the CallbackHandler can not handle the required
callbacks then no chance.
            return false;
        }

        String name = ncb.getName();
        Object credential = ocb.getCredential();

        if (credential instanceof OuterUserCredential) {
            // This credential type will only be seen for a delegation
request, if not seen then the request is not for us.

            if (delegationAcceptable(name, (OuterUserCredential)
credential)) {
                identity = new SimplePrincipal(name);
                if (getUseFirstPass()) {
```

```
                    String userName = identity.getName();
                    if (log.isDebugEnabled())
                        log.debug("Storing username '" + userName + "'
and empty password");
                    // Add the username and an empty password to the
shared state map
                    sharedState.put("javax.security.auth.login.name",
identity);
                    sharedState.put("javax.security.auth.login.password",
"");
                }
                loginOk = true;
                return true;
            }
        }
        return false; // Attempted login but not successful.
    }

    // Make a trust user to decide if the user switch is acceptable.
    protected boolean delegationAcceptable(String requestedUser,
OuterUserCredential connectionUser) {
    if (delegationMappings == null) {
        return false;
    }

    String[] allowedMappings = loadPropertyValue(connectionUser.getName(),
connectionUser.getRealm());
    if (allowedMappings.length == 1 && "*".equals(allowedMappings[0])) {
        // A wild card mapping was found.
        return true;
    }
    for (String current : allowedMappings) {
        if (requestedUser.equals(current)) {
            return true;
        }
    }
    return false;
}
```



NOTE

See the **`ejb-security-interceptors`** quickstart **`README.html`** file for complete instructions and more detailed information about the code examples.

7.5. USE A CLIENT INTERCEPTOR IN AN APPLICATION

An application can insert a client interceptor into the **`EJBClientContext`** interceptor chain either programmatically or by using the service loader mechanism.

Insert the Interceptor Programmatically.
Call the **`org.jboss.ejb.client.EJBClientContext.registerInterceptor(int order, EJBClientInterceptor interceptor)`** method and pass the **`order`** and the **`interceptor`** instance. The **`order`** determines where this client interceptor is placed in the interceptor chain.

Insert the Interceptor Using the Service Loader Mechanism

Create a **`META-INF/services/org.jboss.ejb.client.EJBClientInterceptor`** file and place or package it in the class path of the client application. The rules for the file are dictated by the Java ServiceLoader Mechanism.

- This file is expected to contain a separate line for each fully qualified class name of the EJB client interceptor implementation.
- The EJB client interceptor classes must be available in the class path.

EJB client interceptors that are added using the service loader mechanism are added in the order they are found in the class path and are added to the end of the client interceptor chain. The **`ejb-security-interceptors`** quickstart that ships with Red Hat JBoss Enterprise Application Platform uses this approach.

CHAPTER 8. CLUSTERED ENTERPRISE JAVABEANS

8.1. ABOUT CLUSTERED ENTERPRISE JAVABEANS (EJBS)

EJB components can be clustered for high-availability scenarios. They use different protocols than HTTP components, so they are clustered in different ways. EJB 2 and 3 stateful and stateless beans can be clustered.

For information on singletons, see HA Singleton Service in the JBoss EAP *Development Guide*

8.2. DEPLOYING CLUSTERED EJBS

Clustering support is available in the HA profiles of JBoss EAP 7.0. Starting the standalone server with HA capabilities enabled, involves starting it with the standalone-ha.xml (or even standalone-full-ha.xml):

```
./standalone.sh -server-config=standalone-ha.xml
```

This will start a single instance of the server with HA capabilities.

Obviously, to be able to see the benefits of clustering, you'll need more than one instance of the server. So let's start another server with HA capabilities. That another instance of the server can either be on the same machine or on some other machine. If it's on the same machine, you will need to take care of two things -

- Pass the port offset for the second instance
- Make sure that each of the server instances have a unique **`jboss.node.name`** system property.

You can do that by passing the following two system properties to the startup command:

```
./standalone.sh -server-config=standalone-ha.xml -
Djboss.socket.binding.port-offset=<offset of your choice> -
Djboss.node.name=<unique node name>
```

Follow whichever approach you feel comfortable with for deploying the EJB deployment to this instance too.

WARNING

Deploying the application on just one node of a standalone instance of a clustered server does not mean that it will be automatically deployed to the other clustered instance. You will have to do deploy it explicitly on the other standalone clustered instance too. Or you can start the servers in domain mode so that the deployment can be deployed to all the servers within a server group.

Now that you have deployed an application with clustered EJBs on both the instances, the EJBs are now capable of making use of the clustering features.

NOTE

Starting JBoss EAP 7, if JBoss EAP is started using an HA profile, the state of your SFSBs will be replicated. You no longer need to use the **`@Clustered`** annotation to enable clustering behavior.

Disabling this behavior is achievable on a per-EJB basis by annotating your bean using **`@Stateful(passivationCapable=false)`**, which is new to the EJB 3.2 specification; or globally, via the **`ejb3`** subsystem.

8.3. FAILOVER FOR CLUSTERED EJBS

Clustered EJBs have failover capability. The state of the **`@Stateful`** EJBs is replicated across the cluster nodes so that if one of the nodes in the cluster goes down, some other node will be able to take over the invocations.

8.4. REMOTE STANDALONE CLIENTS

A standalone remote client can use either the JNDI approach or native JBoss EJB client APIs to communicate with the servers. The important thing to note is that when you are invoking clustered EJB deployments, you do not have to list all the servers within the cluster (which obviously wouldn't have been feasible due the dynamic nature of cluster node additions within a cluster).

The remote client has to list only one of the servers with the clustering capability. This server will act as the starting point for cluster topology communication between the client and the clustered nodes.

Note that you have to configure the *ejb* cluster in the **`jboss-ejb-client.properties`** configuration file:

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOU
S=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
```

8.5. CLUSTER TOPOLOGY COMMUNICATION

When a client connects to a server, the JBoss EJB client implementation communicates internally with the server for the cluster topology information, if the server has clustering capability. For example, assuming that server X is listed as the initial server to connect to, when the client connects to server X, the server will send back an asynchronous cluster topology message to the client. This topology message consists of the cluster name and the information of the nodes that belong to the cluster. The node information includes the node address and port number to connect to, when required. So in this example, server X will send back the cluster topology consisting of the other server Y that belongs to the cluster.

In case of stateful clustered EJBs, the invocation flow happens in two steps.

1. Creation of a session for the stateful bean, which happens when you do a JNDI lookup for that bean.

2. Invocation of the returned proxy.

The lookup for the stateful bean, internally, triggers a synchronous session creation request from the client to the server. In this case, the session creation request goes to server X because it was configured

in the **jboss-ejb-client.properties** file. Since server X is clustered, it will return a session id and send back an *affinity* of that session. In case of clustered servers, the *affinity* is equal to the name of the cluster to which the stateful bean belongs on the server side. For non-clustered beans, the affinity is the node name on which the session was created. This *affinity* will help the EJB client to route the invocations on the proxy, as appropriate, to either a node within a cluster for clustered beans, or to a specific node for non-clustered beans. While this session creation request is going on, server X will also send back an asynchronous message that contains the cluster topology. The JBoss EJB client implementation will record this topology information and use it later for connection creation to nodes within the cluster and routing invocations to those nodes, when required.

To understand how failover works, consider the same example of server X being the starting point and a client application looking up a stateful bean and invoking it. During these invocations, the client side collects the cluster topology information from the server. Assuming that for some reason server X goes down and the client application subsequently invokes on the proxy. The JBoss EJB client implementation at this stage must be aware of the *affinity*, and in this case it is the cluster affinity. From the cluster topology information that the client has, it knows that the cluster has two nodes, server X and server Y. When the invocation arrives, the client notices that server X is down, so it uses a selector to fetch a suitable node from the cluster nodes. When the selector returns a node from the cluster nodes, the JBoss EJB client implementation creates a connection to that node, if the connection was not already created earlier, and creates an EJB receiver out of it. Since in this example, the only other node in the cluster is server Y, the selector will return server Y as the node and the JBoss EJB client implementation will use it to create an EJB receiver out of it and use this receiver to pass on the invocation on the proxy. Effectively, the invocation has now failed over to a different node within the cluster.

8.6. REMOTE CLIENTS ON ANOTHER INSTANCE

This section explains how a client application deployed on a JBoss EAP instance invokes a clustered stateful bean that is deployed on another JBoss EAP instance.

In the following example, there are three servers involved. Servers X and Y both belong to a cluster and have clustered EJBs deployed on them. There is another server instance server C, which may or may not have clustering capability. Server C acts as a client on which there is a deployment that wants to invoke the clustered beans deployed on servers X and Y and achieve failover.

The configurations are done in the **jboss-ejb-client.xml** file, which points to a remote outbound connection to the other server. The configuration in the **jboss-ejb-client.xml** file is in the deployment of server C because server C is the client. The client configuration need not point to all the clustered nodes, but just to one of them. This will act as a starting point for the communication.

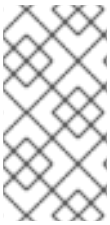
In this case, a remote outbound connection is created from server C to server X and then server X is used as the starting point for the communication. Similar to the case of remote standalone clients, when the application on server C looks up a stateful bean, a session creation request is sent to server X that returns a session id and the cluster affinity for it. Server X also sends back an asynchronous message to server C containing the cluster topology. This topology information includes the node information of server Y, because server Y belongs to the cluster along with server X. Subsequent invocations on the proxy will be routed appropriately to the nodes in the cluster. If server X goes down, as explained earlier, a different node from the cluster will be selected and the invocation will be forwarded to that node.

Both remote standalone clients as well as remote clients on another JBoss EAP instance act similarly in terms of failover.

8.7. STANDALONE AND IN-SERVER CLIENT CONFIGURATION

To connect an EJB client to a clustered EJB application, you need to expand the existing configuration in standalone EJB client or in-server EJB client to include cluster connection configuration. The **jboss-**

ejb-client.properties for standalone EJB client, or even **jboss-ejb-client.xml** file for a server-side application must be expanded to include a cluster configuration.



NOTE

An EJB client is any program that uses an EJB on a remote server. A client is **in-server** when the EJB client calling the remote server is itself running inside of a server. In other words, a JBoss EAP instance calling out to another JBoss EAP instance would be considered an in-server client.

This example shows the additional cluster configuration required for a standalone EJB client.

```
remote.clusters=ejb
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOU
S=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
remote.cluster.ejb.username=test
remote.cluster.ejb.password=password
```

If an application uses the remote-outbound-connection, you need to configure the **jboss-ejb-client.xml** file and add cluster configuration as shown in the following example:

```
<jboss-ejb-client xmlns:xsi="urn:jboss:ejb-client:1.2"
xsi:noNamespaceSchemaLocation="jboss-ejb-client_1_2.xsd">
  <client-context>
    <ejb-receivers>
      <!-- this is the connection to access the app-one -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-
connection-1" />
      <!-- this is the connection to access the app-two -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-
connection-2" />
    </ejb-receivers>

    <!-- If an outbound connection connects to a cluster,
             a list of members is provided after successful connection.
         To connect to this node this cluster element must be defined. -->

    <clusters>
      <!-- cluster of remote-ejb-connection-1 -->
      <cluster name="ejb" security-realm="ejb-security-realm-1"
username="quickuser1">
        <connection-creation-options>
        <property name="org.xnio.Options.SSL_ENABLED" value="false" />
        <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS"
value="false" />
        </connection-creation-options>
      </cluster>
    </clusters>
  </client-context>
</jboss-ejb-client>
```

For more information about remote-outbound-connection, see About the Remoting Subsystem in the JBoss EAP *Configuration Guide*.

NOTE

For a secure connection you need to add the credentials to cluster configuration in order to avoid an authentication exception.

8.8. IMPLEMENTING A CUSTOM LOAD BALANCING POLICY FOR EJB CALLS

It is possible to implement an alternate or customized load balancing policy in order to balance an application's EJB calls across servers.

You can implement **`AllClusterNodeSelector`** for EJB calls. The node selection behavior of **`AllClusterNodeSelector`** is similar to default selector except that **`AllClusterNodeSelector`** uses all available cluster nodes even in case of a large cluster (number of nodes > 20). If an unconnected cluster node is returned, it is opened automatically. The following example shows **`AllClusterNodeSelector`** implementation:

```
package org.jboss.as.quickstarts.ejb.clients.selector;

import java.util.Arrays;
import java.util.Random;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.jboss.ejb.client.ClusterNodeSelector;
public class AllClusterNodeSelector implements ClusterNodeSelector {
  private static final Logger LOGGER =
Logger.getLogger(AllClusterNodeSelector.class.getName());

  @Override
  public String selectNode(final String clusterName, final String[]
connectedNodes, final String[] availableNodes) {
    if(LOGGER.isLoggable(Level.FINER)) {
      LOGGER.finer("INSTANCE "+this+ " : cluster:"+clusterName+"
connected:"+Arrays.deepToString(connectedNodes)+"
available:"+Arrays.deepToString(availableNodes));
    }

    if (availableNodes.length == 1) {
        return availableNodes[0];
    }
    final Random random = new Random();
    final int randomSelection = random.nextInt(availableNodes.length);
    return availableNodes[randomSelection];
  }

}
```

You can also implement the **`SimpleLoadFactorNodeSelector`** for EJB calls. Load balancing in **`SimpleLoadFactorNodeSelector`** happens based on a load factor. The load factor (2/3/4) is calculated based on the names of nodes (A/B/C) irrespective of the load on each node. The following example shows **`SimpleLoadFactorNodeSelector`** implementation:

```
package org.jboss.as.quickstarts.ejb.clients.selector;
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.logging.Level;
import java.util.logging.Logger;

import org.jboss.ejb.client.DeploymentNodeSelector;
public class SimpleLoadFactorNodeSelector implements
DeploymentNodeSelector {
  private static final Logger LOGGER =
Logger.getLogger(SimpleLoadFactorNodeSelector.class.getName());
  private final Map<String, List<String>[]> nodes = new HashMap<String,
List<String>[]>();
  private final Map<String, Integer> cursor = new HashMap<String, Integer>
();

  private ArrayList<String> calculateNodes(Collection<String>
eligibleNodes) {
    ArrayList<String> nodeList = new ArrayList<String>();

    for (String string : eligibleNodes) {
      if(string.contains("A") || string.contains("2")) {
        nodeList.add(string);
        nodeList.add(string);
      } else if(string.contains("B") || string.contains("3")) {
        nodeList.add(string);
        nodeList.add(string);
        nodeList.add(string);
      } else if(string.contains("C") || string.contains("4")) {
        nodeList.add(string);
        nodeList.add(string);
        nodeList.add(string);
        nodeList.add(string);
      }
    }
    return nodeList;
  }

  @SuppressWarnings("unchecked")
  private void checkNodeNames(String[] eligibleNodes, String key) {
    if(!nodes.containsKey(key) || nodes.get(key)[0].size() !=
eligibleNodes.length || !nodes.get(key)
[0].containsAll(Arrays.asList(eligibleNodes))) {
      // must be synchronized as the client might call it concurrent
      synchronized (nodes) {
        if(!nodes.containsKey(key) || nodes.get(key)[0].size() !=
eligibleNodes.length || !nodes.get(key)
[0].containsAll(Arrays.asList(eligibleNodes))) {
          ArrayList<String> nodeList = new ArrayList<String>();
          nodeList.addAll(Arrays.asList(eligibleNodes));

          nodes.put(key, new List[] { nodeList, calculateNodes(nodeList)
```

```
});
        }
      }
    }
  }
   private synchronized String nextNode(String key) {
    Integer c = cursor.get(key);
    List<String> nodeList = nodes.get(key)[1];

    if(c == null || c >= nodeList.size()) {
      c = Integer.valueOf(0);
    }

    String node = nodeList.get(c);
    cursor.put(key, Integer.valueOf(c + 1));

    return node;
  }

  @Override
  public String selectNode(String[] eligibleNodes, String appName, String
moduleName, String distinctName) {
    if (LOGGER.isLoggable(Level.FINER)) {
      LOGGER.finer("INSTANCE " + this + " : nodes:" +
Arrays.deepToString(eligibleNodes) + " appName:" + appName + "
moduleName:" + moduleName
          + " distinctName:" + distinctName);
    }

    // if there is only one there is no sense to choice
    if (eligibleNodes.length == 1) {
      return eligibleNodes[0];
    }
    final String key = appName + "|" + moduleName + "|" + distinctName;

    checkNodeNames(eligibleNodes, key);
    return nextNode(key);
  }
}
```

Configuring the jboss-ejb-client.properties File

You need to add the property **`remote.cluster.ejb.clusternode.selector`** with the name of your implementation class (**`AllClusterNodeSelector`** or **`SimpleLoadFactorNodeSelector`**). The selector will see all configured servers that are available at the invocation time. The following example uses **`AllClusterNodeSelector`** as the cluster node selector:

```
remote.clusters=ejb
remote.cluster.ejb.clusternode.selector=org.jboss.as.quickstarts.ejb.clien
ts.selector.AllClusterNodeSelector
remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOANONYMOU
S=false
remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED=false
remote.cluster.ejb.username=test
remote.cluster.ejb.password=password

remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABLED=fals
```

```
e
remote.connections=one,two
remote.connection.one.host=localhost
remote.connection.one.port = 8080
remote.connection.one.connect.options.org.xnio.Options.SASL_POLICY_NOANONY
MOUS=false
remote.connection.one.username=user
remote.connection.one.password=user123
remote.connection.two.host=localhost
remote.connection.two.port = 8180
remote.connection.two.connect.options.org.xnio.Options.SASL_POLICY_NOANONY
MOUS=false
```

Using EJB Client API

You need to add the property **`remote.cluster.ejb.clusternode.selector`** to the list for the **`PropertiesBasedEJBClientConfiguration`** constructor. The following example uses **`AllClusterNodeSelector`** as the cluster node selector:

```
Properties p = new Properties();
p.put("remote.clusters", "ejb");
p.put("remote.cluster.ejb.clusternode.selector",
"org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSelector");
p.put("remote.cluster.ejb.connect.options.org.xnio.Options.SASL_POLICY_NOA
NONYMOUS", "false");
p.put("remote.cluster.ejb.connect.options.org.xnio.Options.SSL_ENABLED",
"false");
p.put("remote.cluster.ejb.username", "test");
p.put("remote.cluster.ejb.password", "password");

p.put("remote.connectionprovider.create.options.org.xnio.Options.SSL_ENABL
ED", "false");
p.put("remote.connections", "one,two");
p.put("remote.connection.one.port", "8080");
p.put("remote.connection.one.host", "localhost");
p.put("remote.connection.two.port", "8180");
p.put("remote.connection.two.host", "localhost");

EJBClientConfiguration cc = new PropertiesBasedEJBClientConfiguration(p);
ContextSelector<EJBClientContext> selector = new
ConfigBasedEJBClientContextSelector(cc);
EJBClientContext.setSelector(selector);

p = new Properties();
p.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
InitialContext context = new InitialContext(p);
```

Configuring the jboss-ejb-client.xml File

To use the load balancing policy for server to server communication, package the class together with the application and configure it within the **`jboss-ejb-client.xml`** settings located in **`META-INF`** folder. The following example uses **`AllClusterNodeSelector`** as the cluster node selector:

```
<jboss-ejb-client xmlns:xsi="urn:jboss:ejb-client:1.2"
xsi:noNamespaceSchemaLocation="jboss-ejb-client_1_2.xsd">
  <client-context deployment-node-
selector="org.jboss.ejb.client.DeploymentNodeSelector">
```

```
    <ejb-receivers>
      <!-- This is the connection to access the application. -->
      <remoting-ejb-receiver outbound-connection-ref="remote-ejb-
connection-1" />
    </ejb-receivers>
    <!-- Specify the cluster configurations applicable for this client
context -->
    <clusters>
      <!-- Configure the cluster of remote-ejb-connection-1. -->
      <cluster name="ejb" security-realm="ejb-security-realm-1"
username="test" cluster-node-
selector="org.jboss.as.quickstarts.ejb.clients.selector.AllClusterNodeSele
ctor">
        <connection-creation-options>
          <property name="org.xnio.Options.SSL_ENABLED" value="false" />
          <property name="org.xnio.Options.SASL_POLICY_NOANONYMOUS"
value="false" />
        </connection-creation-options>
      </cluster>
    </clusters>
  </client-context>
</jboss-ejb-client>
```

To use the above configuration with security, you will need to add **`ejb-security-realm-1`** to client-server configuration. The following example shows the CLI commands for adding security realm (**`ejb-security-realm-1`**) the value is the base64 encoded password for the user "test":

```
core-service=management/security-realm=ejb-security-realm-1:add()
core-service=management/security-realm=ejb-security-realm-1/server-
identity=secret:add(value=cXVpY2sxMjMr)
```

If the load balancing policy should be used for server to server communication, the class can be packaged together with the application or as a module. This class is configured in the **`jboss-ejb-client`** settings file located in the **`META-INF`** directory of the top-level EAR archive. The following example uses **`RoundRobinNodeSelector`** as the deployment node selector.

```
<jboss-ejb-client xmlns="urn:jboss:ejb-client:1.2">
    <client-context deployment-node-
selector="org.jboss.example.RoundRobinNodeSelector">
        <ejb-receivers>
            <remoting-ejb-receiver outbound-connection-ref="..."/>
        </ejb-receivers>
        ...
    </client-context>
</jboss-ejb-client>
```

NOTE

If you are running a standalone server, use the start option **`-Djboss.node.name=`** or the server configuration file **`standalone.xml`** to configure the server name. Ensure that the server name is unique. If you are running a managed domain, the host controller automatically validates that the names are unique.

APPENDIX A. REFERENCE MATERIAL

A.1. EJB JNDI NAMING REFERENCE

The JNDI lookup name for a session bean uses the following syntax:

```
ejb:<appName>/<moduleName>/<distinctName>/<beanName>!<viewClassName>?
stateful
```

- **<appName>**: If the session bean's JAR file has been deployed within an enterprise archive (EAR) then the **appName** is the name of the respective EAR. By default, the name of an EAR is its filename without the **.ear** suffix. The application name can be overridden in its **application.xml** file. If the session bean is not deployed in an EAR, then leave the **appName** blank.
- **<moduleName>**: The **moduleName** is the name of the JAR file in which the session bean is deployed. The default name of the JAR file is its filename without the **.jar** suffix. The module name can be overridden in the JAR's **ejb-jar.xml** file.
- **<distinctName>**: JBoss EAP allows each deployment to specify an optional distinct name. If the deployment does not have a distinct name, then leave the **distinctName** blank.
- **<beanName>**: The **beanName** is the simple class name of the session bean to be invoked.
- **<viewClassName>**: The **viewClassName** is the fully qualified class name of the remote interface. This includes the package name of the interface.
- **?stateful**: The **?stateful** suffix is required when the JNDI name refers to a stateful session bean. It is not included for other bean types.

For example, if we deployed **hello.jar** having a stateful bean **org.jboss.example.HelloBean** that exposed a remote interface **org.jboss.example.Hello**, then the JNDI lookup name would be:

```
ejb:/hello/HelloBean!org.jboss.example.Hello?stateful"
```

A.2. EJB REFERENCE RESOLUTION

This section covers how JBoss EAP implements **@EJB** and **@Resource**. Please note that XML always overrides annotations but the same rules apply.

Rules for the @EJB annotation

- The **@EJB** annotation also has a **mappedName()** attribute. The specification leaves this as vendor specific metadata, but JBoss EAP 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;
  ```

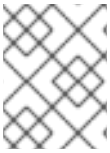
  Then the following rules apply:

- The EJB jar of the referencing bean is searched for an EJB with the interface used in the **@EJB** injection. If there are more than one EJB that publishes same business interface, then an exception is thrown. If there is only one bean with that interface then that one is used.
- Search the EAR for EJBs that publish that interface. If there are duplicates, then an exception is thrown. Otherwise the matching bean is returned.
- Search globally in JBoss EAP runtime for an EJB of that interface. Again, if duplicates are found, an exception is thrown.

- **@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: it allows you to put a relative path to a jar in the EAR where the EJB you are referencing is located. Refer to the EJB 3.2 specification for more details.

A.3. PROJECT DEPENDENCIES FOR REMOTE EJB CLIENTS

Maven projects that include the invocation of session beans from remote clients require the following dependencies from the JBoss EAP Maven repository.



NOTE

The **artifactId** versions are subject to change. Refer to the JBoss EAP Maven Repository for the latest versions.

Table A.1. Maven Dependencies for Remote EJB Clients

GroupID	ArtifactID
org.jboss.bom	jboss-eap-javaee7
org.jboss.spec.javax.transaction	jboss-transaction-api_1.2_spec
org.jboss.spec.javax.ejb	jboss-ejb-api_3.2_spec
org.jboss.eap	wildfly-ejb-client-bom

The **jboss-eap-javaee7** "Bill of Materials" (BOM) packages the correct version of many of the artifacts commonly required by a JBoss EAP application. The BOM dependency is specified in the **<dependencyManagement>** section of the **pom.xml** with the scope of **import**.

Example POM File <dependencyManagement> Section

```
<dependencyManagement>
   <dependencies>
      <dependency>
      <groupId>org.jboss.bom</groupId>
        <artifactId>jboss-eap-javaee7</artifactId>
        <version>${version.jboss.bom.eap}</version>
```

```
        <type>pom</type>
        <scope>import</scope>
      </dependency>
   </dependencies>
</dependencyManagement>
```

The remaining dependencies are specified in the **<dependencies>** section of the **pom.xml** file with a scope of **runtime**.

Example POM File <dependencies> Section

```
<dependencies>
    <!-- Include the EJB client JARs -->
    <dependency>
        <groupId>org.jboss.eap</groupId>
        <artifactId>wildfly-ejb-client-bom</artifactId>
        <type>pom</type>
        <scope>compile</scope>
    </dependency>

    <!-- Include any additional dependencies required by the application
        ...
    -->

</dependencies>
```

The **ejb-remote** quickstart that ships with JBoss EAP provides a complete working example of remote EJB client application. See the **client/pom.xml** file located in root directory of that quickstart for a complete example of dependency configuration for remote session bean invocation.

A.4. JBOSS-EJB3.XML DEPLOYMENT DESCRIPTOR REFERENCE

jboss-ejb3.xml is a custom deployment descriptor that can be used in either EJB JAR or WAR archives. In an EJB JAR archive it must be located in the **META-INF/** directory. In a WAR archive it must be located in the **WEB-INF/** directory.

The format is similar to **ejb-jar.xml**, using some of the same namespaces and providing some other additional namespaces. The contents of **jboss-ejb3.xml** are merged with the contents of **ejb-jar.xml**, with the **jboss-ejb3.xml** items taking precedence.

This document only covers the additional non-standard namespaces used by **jboss-ejb3.xml**. Refer to http://java.sun.com/xml/ns/javaee/ for documentation on the standard namespaces.

The root namespace is http://www.jboss.com/xml/ns/javaee.

Assembly descriptor namespaces

The following namespaces can all be used in the **<assembly-descriptor>** element. They can be used to apply their configuration to a single bean, or to all beans in the deployment by using * as the ejb-name.

The clustering namespace: urn:clustering:1.0

```
xmlns:c="urn:clustering:1.0"
```

This allows you to mark EJB's as clustered. It is the deployment descriptor equivalent to **`@org.jboss.ejb3.annotation.Clustered`**.

```
<c:clustering>
   <ejb-name>DDBasedClusteredSFSB</ejb-name>
   <c:clustered>true</c:clustered>
</c:clustering>
```

The security namespace (`urn:security`)

```
xmlns:s="urn:security"
```

This allows you to set the **`security-domain`** and the **`run-as-principal`** for an EJB.

```
<s:security>
  <ejb-name>*</ejb-name>
  <s:security-domain>myDomain</s:security-domain>
  <s:run-as-principal>myPrincipal</s:run-as-principal>
</s:security>
```

The resource adapter namespace: `urn:resource-adapter-binding`

```
xmlns:r="urn:resource-adapter-binding"
```

This allows you to set the resource adapter for a Message-Driven Bean.

```
<r:resource-adapter-binding>
  <ejb-name>*</ejb-name>
  <r:resource-adapter-name>myResourceAdapter</r:resource-adapter-name>
</r:resource-adapter-binding>
```

The IIOP namespace: `urn:iiop`

```
xmlns:u="urn:iiop"
```

The IIOP namespace is where IIOP settings are configured.

The pool namespace: `urn:ejb-pool:1.0`

```
xmlns:p="urn:ejb-pool:1.0"
```

This allows you to select the pool that is used by the included stateless session beans or Message-Driven Beans. Pools are defined in the server configuration.

```
<p:pool>
   <ejb-name>*</ejb-name>
   <p:bean-instance-pool-ref>my-pool</p:bean-instance-pool-ref>
</p:pool>
```

The cache namespace: `urn:ejb-cache:1.0`

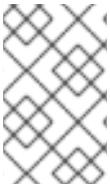
-

```
xmlns:c="urn:ejb-cache:1.0"
```

This allows you to select the cache that is used by the included stateful session beans. Caches are defined in the server configuration.

```
<c:cache>
   <ejb-name>*</ejb-name>
   <c:cache-ref>my-cache</c:cache-ref>
</c:cache>
```

```
<?xml version="1.1" encoding="UTF-8"?>
<jboss:ejb-jar xmlns:jboss="http://www.jboss.com/xml/ns/javaee"
    xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:c="urn:clustering:1.0"
    xsi:schemaLocation="http://www.jboss.com/xml/ns/javaee
http://www.jboss.org/j2ee/schema/jboss-ejb3-2_0.xsd"
    version="3.1"
    impl-version="2.0">
  <enterprise-beans>
     <message-driven>
        <ejb-name>ReplyingMDB</ejb-name>
        <ejb-
class>org.jboss.as.test.integration.ejb.mdb.messagedestination.ReplyingM
DB</ejb-class>
        <activation-config>
           <activation-config-property>
              <activation-config-property-name>destination</activation-
config-property-name>
              <activation-config-property-
value>java:jboss/mdbtest/messageDestinationQueue
              </activation-config-property-value>
           </activation-config-property>
        </activation-config>
     </message-driven>
  </enterprise-beans>
  <assembly-descriptor>
     <c:clustering>
        <ejb-name>DDBasedClusteredSFSB</ejb-name>
        <c:clustered>true</c:clustered>
     </c:clustering>
  </assembly-descriptor>
</jboss:ejb-jar>
```



NOTE

There are known issues with the **`jboss-ejb3-spec-2_0.xsd`** file that may result in schema validation errors. You can ignore these errors. For more information, see https://bugzilla.redhat.com/show_bug.cgi?id=1192591.

A.5. CONFIGURE AN EJB THREAD POOL

You can create an EJB Thread pool using the management console or the management CLI.

Configure an EJB Thread Pool Using the Management Console

1. Log in to the management console.
2. Click on the **Configuration** tab. Expand the **Subsystems** menu.
3. Select **EJB 3** and then click **View**.
4. Select the **Container** tab and then click **Thread Pools**.
5. Click **Add**. The **Create THREAD-POOL** dialog appears.
6. Specify the required details, **Name** and **Max threads** value.
7. Click **Save**.

Configure an EJB Thread Pool Using the Management CLI

Use the **add** operation with the following syntax:

```
/subsystem=ejb3/thread-pool=THREADPOOLNAME:add(max-threads=MAXSIZE)
```

- Replace **THREADPOOLNAME** with the required name for the thread pool.
- Replace **MAXSIZE** with the maximum size of the thread pool.

Use the **read-resource** operation to confirm the creation of the bean pool:

```
/subsystem=ejb3/thread-pool=THREADPOOLNAME:read-resource
```

To reconfigure all the services in the **ejb3** subsystem to use a new thread pool, use the following commands:

```
/subsystem=ejb3/thread-pool=bigger:add(max-threads=100)
/subsystem=ejb3/service=async:write-attribute(name=thread-pool-name,
value="bigger")
/subsystem=ejb3/service=remote:write-attribute(name=thread-pool-name,
value="bigger")
/subsystem=ejb3/service=timer-service:write-attribute(name=thread-pool-
name, value="bigger")
reload
```

XML Configuration Sample:

```
<subsystem xmlns="urn:jboss:domain:ejb3:4.0">
    ...
    <async thread-pool-name="bigger"/>
    ...
    <timer-service thread-pool-name="bigger" default-data-store="default-
file-store">
    ...
    <remote connector-ref="http-remoting-connector" thread-pool-
name="bigger"/>
    ...
    <thread-pools>
        <thread-pool name="default">
```

```
            <max-threads count="10"/>
            <keepalive-time time="100" unit="milliseconds"/>
        </thread-pool>
        <thread-pool name="bigger">
            <max-threads count="100"/>
        </thread-pool>
    </thread-pools>
    ...
```

NOTE

`keepalive-time` should not be used as it is not effective.

Revised on 2018-02-08 10:17:17 EST