JBoss Enterprise Application Platform Common Criteria Certification 5

JBoss Microcontainer User Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Last Updated: 2017-11-20

JBoss Enterprise Application Platform Common Criteria Certification 5
JBoss Microcontainer User Guide

for use with JBoss Enterprise Application Platform 5 Common Criteria Certification
Edition 5.1.0

Mark Newton
Red Hat
mark.newton@jboss.org

Aleš Justin
Red Hat
ajustin@redhat.com

Edited by

Misty Stanley-Jones
Red Hat
misty@redhat.com

Legal Notice

Copyright © 2011 Red Hat, Inc.

This document is licensed by Red Hat under the Creative Commons Attribution-ShareAlike 3.0 Unported License. If you distribute this document, or a modified version of it, you must provide attribution to Red Hat, Inc. and provide a link to the original. If the document is modified, all Red Hat trademarks must be removed.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide is intended for Java developers who wish to use the JBoss Microcontainer to deploy customized, modular Java environments for their applications.

Table of Contents

PART I. INTRODUCTION TO THE MICROCONTAINER - GUIDED TUTORIAL 4

CHAPTER 1. PREREQUISITES TO USING THIS GUIDE 5
1.1. INSTALL MAVEN 5
1.2. SPECIAL MAVEN SETTINGS FOR THE MICROCONTAINER EXAMPLES 8
1.3. DOWNLOADING THE EXAMPLES 9

CHAPTER 2. INTRODUCTION TO THE MICROCONTAINER 10
2.1. FEATURES 10
2.2. DEFINITIONS 10
2.3. INSTALLATION 11

CHAPTER 3. BUILDING SERVICES 12
3.1. INTRODUCTION TO THE HUMAN RESOURCES EXAMPLE 12
3.2. COMPILING THE HRMANAGER EXAMPLE PROJECT 13
3.3. CREATING POJOS 13
3.3.1. XML Deployment Descriptors 13
3.4. CONNECTING POJOS TOGETHER 13
3.4.1. Special Considerations 14
3.5. WORKING WITH SERVICES 14
3.5.1. Configuring A Service 14
3.5.2. Testing A Service 15
3.5.3. Packaging A Service 17

CHAPTER 4. USING SERVICES 19
4.1. BOOTSTRAPPING THE MICROCONTAINER 22
4.2. DEPLOYING THE SERVICE 23
4.3. DIRECT ACCESS 24
4.4. INDIRECT ACCESS 26
4.5. DYNAMIC CLASSLOADING 27
4.5.1. Problems With Classloaders Created with Deployment Descriptors 31

CHAPTER 5. ADDING BEHAVIOR WITH AOP 32
5.1. CREATING AN ASPECT 32
5.2. CONFIGURING THE MICROCONTAINER FOR AOP 34
5.3. APPLYING AN ASPECT 35
5.4. LIFECYCLE CALLBACKS 37
5.5. ADDING SERVICE LOOK-UPS THROUGH JNDI 39

PART II. ADVANCED CONCEPTS WITH THE MICROCONTAINER 41

CHAPTER 6. COMPONENT MODELS 42
6.1. ALLOWABLE INTERACTIONS WITH COMPONENT MODELS 42
6.2. A BEAN WITH NO DEPENDENCIES 42
6.3. USING THE MICROCONTAINER WITH SPRING 42
6.4. USING GUICE WITH THE MICROCONTAINER 43
6.5. LEGACY MBEANS, AND MIXING DIFFERENT COMPONENT MODELS 45
6.6. EXPOSING POJOS AS MBEANS 46

CHAPTER 7. ADVANCED DEPENDENCY INJECTION AND IOC 50
7.1. VALUE FACTORY 50
7.2. CALLBACKS 51
7.3. BEAN ACCESS MODE 53
7.4. BEAN ALIAS 54

7.5. XML (OR METADATA) ANNOTATIONS SUPPORT 55
7.6. AUTOWIRE 57
7.7. BEAN FACTORY 57
7.8. BEAN METADATA BUILDER 59
7.9. CUSTOM CLASSLOADER 60
7.10. CONTROLLER MODE 61
7.11. CYCLE 62
7.12. DEMAND AND SUPPLY 63
7.13. INSTALLS 63
7.14. LAZY MOCK 64
7.15. LIFECYCLE 64

CHAPTER 8. THE VIRTUAL FILE SYSTEM 66
8.1. VFS PUBLIC API 67
8.2. VFS ARCHITECTURE 74
8.3. EXISTING IMPLEMENTATIONS 74
8.4. EXTENSION HOOKS 75
8.5. FEATURES 76

CHAPTER 9. THE CLASSLOADING LAYER 77
9.1. CLASSLOADER 77
9.2. CLASSLOADING 83
9.3. CLASSLOADING VFS 88

CHAPTER 10. THE VIRTUAL DEPLOYMENT FRAMEWORK 90
10.1. AGNOSTIC HANDLING OF DEPLOYMENT TYPES 90
10.2. SEPARATION OF STRUCTURE RECOGNITION FROM DEPLOYMENT LIFECYCLE LOGIC 90
10.3. NATURAL FLOW CONTROL IN THE FORM OF ATTACHMENTS 93
10.4. CLIENT, USER, AND SERVER USAGE AND IMPLEMENTATION DETAILS 94
10.5. SINGLE STATE MACHINE 94
10.6. SCANNING CLASSES FOR ANNOTATIONS 95

APPENDIX A. REVISION HISTORY 96

PART I. INTRODUCTION TO THE MICROCONTAINER - GUIDED TUTORIAL

CHAPTER 1. PREREQUISITES TO USING THIS GUIDE

To use the examples in this guide, you need to install and configure some supporting software, and download the code for the examples.

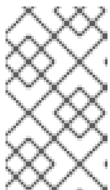
1.1. INSTALL MAVEN

The examples used in this project require **Maven** v2.2.0 or later. Download **Maven** directly from the Apache Maven homepage, and install and configure your system as described in Procedure 1.1, “Install Maven”.

Procedure 1.1. Install Maven

1. **Verify Java Developer Kit 1.6 or above is installed. This is also a requirement for the Enterprise Platform.**
   Ensure you have **Java** installed on your system, and have set the **`JAVA_HOME`** environment variable in your **`~/.bash_profile`** for Linux, or in the System Properties for Windows. For more information regarding setting environment variables, refer to the Step 4 step in this procedure.

2. **Download Maven**

   NOTE

   This step and future steps assume that you have saved Maven to the suggested location for your operating system. Maven, as any other Java application, is able to be installed in any reasonable location on your system.

   Visit http://maven.apache.org/download.html.

   Click the compiled zip archive link, for example **`apache-maven-2.2.1-bin.zip`**

   Select a download mirror from the list.

   For Linux Users

   Save the zip archive to your **home** directory.

   For Windows Users

   Save the zip archive to your **`C:\Documents and Settings\`*`user_name`*** directory.

3. **Install Maven**

   For Linux Users

   Extract the zip file to your **home** directory. If you selected the zip archive in Step 2, and do not rename the directory, the extracted directory is named **`apache-maven-`*`version`***.

   For Windows Users

   Extract the zip archive to C:\Program Files\Apache Software Foundation. If you selected the zip archive in Step 2, and do not rename the directory, the extracted directory is named **`apache-maven-`*`version`***.

4. **Configure Environment Variables**

For Linux Users

Add the following lines to your **~/.bash_profile**. Ensure you change the *[username]* to your actual username, and that the Maven directory is the actual directory name. The version number may be different than the one listed below.

```
export M2_HOME=/home/[username]/apache-maven-2.2.1 export
M2=$M2_HOME/bin export
   PATH=$M2:$PATH
```

By including **M2** at the beginning of your path, the **Maven** version you just installed will be the default version used. You may also want to set the path of your **JAVA_HOME** environment variable to the location of the JDK on your system.

For Windows Users

Add the **M2_HOME**, **M2**, and **JAVA_HOME** environment variables.

1. Press **Start**+**Pause|Break**. The System Properties dialog box is displayed.
2. Click the **Advanced** tab, then click the **Environment Variables** button.
3. Under **System Variables**, select **Path**.
4. Click **Edit**, and append the two **Maven** paths using a semi-colon to separate each entry. Quotation marks are not required around paths.
   - Add the variable **M2_HOME** and set the path to **C:\Program Files\Apache Software Foundation\apache-maven-2.2.1**.
   - Add the variable **M2** and set the value to **%M2_HOME%\bin**.
5. In the same dialog, create the **JAVA_HOME** environment variable:
   - Add the variable **%JAVA_HOME%** and set the value to the location of your JDK. For example **C:\Program Files\Java\jdk1.6.0_02**.
6. In the same dialog, update or create the Path environment variable:
   - Add the variable **%M2%** to allow Maven to be executed from the command-line.
   - Add the variable **%JAVA_HOME%\bin** to set the path to the correct Java installation.
7. Click **OK** until the **System Properties** dialog box closes.

5. **Implement changes to .bash_profile**

   For Linux Users Only

   To update the changes made to the .bash_profile in the current terminal session, source your .bash_profile.

   ```
   [localhost]$ source ~/.bash_profile
   ```

6. **Update gnome-terminal profile**

   For Linux Users Only

Update the terminal profile to ensure that subsequent iterations of gnome-terminal (or Konsole terminal) read the new environment variables.

1. Click **Edit** → **Profiles**
2. Select **Default**, and click the **`Edit`** button.
3. In the **`Editing Profile`** dialog, click the **`Title and Command`** tab.
4. Select the **`Run command as login shell`** check box.
5. Close all open Terminal dialog boxes.

7. **Verify the environment variable changes and Maven install**

For Linux Users

To verify that the changes have been implemented correctly, open a terminal and execute the following commands:

- Execute **`echo $M2_HOME`**, which should return the following result.

```
[localhost]$ echo $M2_HOME /home/username/apache-maven-2.2.1
```

- Execute **`echo $M2`**, which should return the following result.

```
[localhost]$ echo $M2 /home/username/apache-maven-2.2.1/bin
```

- Execute **`echo $PATH`**, and verify the **Maven `/bin`** directory is included.

```
[localhost]$ echo $PATH /home/username/apache-maven-2.2.1/bin
```

- Execute **`which mvn`**, which should display the path to the **Maven** executable.

```
[localhost]$ which mvn ~/apache-maven-2.2.1/bin/mvn
```

- Execute **`mvn -version`**, which should display the **Maven** version, related Java version, and operating system information.

```
[localhost]$ $ mvn -version Apache Maven 2.2.1 (r801777; 2009-08-
07 05:16:01+1000) Java version:
       1.6.0_0 Java home: /usr/lib/jvm/java-1.6.0-openjdk-
1.6.0.0/jre Default locale: en_US, platform encoding: UTF-8 OS
       name: "Linux" version: "2.6.30.9-96.fc11.i586" arch:
"i386" Family:
       "unix"
```

For Windows Users

To verify that the changes have been implemented correctly, open a terminal and execute the following command:

- In a command prompt, execute **`mvn -version`**

```
C:\> mvn -version Apache
       Maven 2.2.1 (r801777; 2009-08-06 12:16:01-0700) Java
version: 1.6.0_17 Java home: C:\Sun\SDK\jdk\jre Default
       locale: en_US, platform encoding: Cp1252 OS name: "windows
xp" version: "5.1" arch:
       "x86" Family: "windows"
```

You have now successfully configured **Maven** for use with the examples in this guide.

1.2. SPECIAL MAVEN SETTINGS FOR THE MICROCONTAINER EXAMPLES

Maven is a modular build system which pulls in dependencies as needed. The examples in this guide assume that you have included the block of XML in Example 1.1, “Example **settings.xml** File” in your **~/.m2/settings.xml** (Linux) or **C:\Documents and Settings*username*\.m2\settings.xml** (Windows). If the file does not exist, you can create it first.

Example 1.1. Example settings.xml File

```
<settings>
 <profiles>
   <profile>
     <id>jboss.repository</id>
     <activation>
       <property>
         <name>!jboss.repository.off</name>
       </property>
     </activation>
     <repositories>
       <repository>
         <id>snapshots.jboss.org</id>
         <url>http://snapshots.jboss.org/maven2</url>
         <snapshots>
           <enabled>true</enabled>
         </snapshots>
       </repository>
       <repository>
         <id>repository.jboss.org</id>
         <url>http://repository.jboss.org/maven2</url>
         <snapshots>
           <enabled>false</enabled>
         </snapshots>
       </repository>
     </repositories>
     <pluginRepositories>
       <pluginRepository>
         <id>repository.jboss.org</id>
         <url>http://repository.jboss.org/maven2</url>
         <snapshots>
           <enabled>false</enabled>
         </snapshots>
       </pluginRepository>
       <pluginRepository>
         <id>snapshots.jboss.org</id>
```

```
          <url>http://snapshots.jboss.org/maven2</url>
          <snapshots>
            <enabled>true</enabled>
          </snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile>
  </profiles>
</settings>
```

1.3. DOWNLOADING THE EXAMPLES

The examples in this guide show you how to create a maven project that depends on the JBoss Microcontainer, using Maven. You can download them from images/examples.zip . This location is subject to change, but is included for expediency.

After you have downloaded the ZIP file containing the examples, extract it to a convenient location and look over the examples to familiarize yourself with their structure.

CHAPTER 2. INTRODUCTION TO THE MICROCONTAINER

The JBoss Microcontainer is a refactoring of the JBoss JMX Microkernel to support direct POJO deployment and standalone use outside the JBoss application server.

The Microcontainer is designed to meet specific needs of Java developers who want to use object-oriented programming techniques to rapidly deploy software. In addition, it allows software to be deployed on a wide range of devices, from mobile computing platforms, large-scale grid-computing environments, and everything in between.

2.1. FEATURES

- All the features of the JMX Microkernel
- Direct POJO deployment (no need for Standard/XMBean or MBeanProxy)
- Direct IOC style dependency injection
- Improved lifecycle management
- Additional control over dependencies
- Transparent AOP integration
- Virtual File System
- Virtual Deployment Framework
- OSGi classloading

2.2. DEFINITIONS

This guide uses some terms that may not be familiar. Some of them are defined in Microcontainer Definition List.

Microcontainer Definition List

JMX Microkernel

The JBoss JMX Microkernel is a modular Java environment. It differs from standard environments like J2EE in that the developer is able to choose exactly which components are part of the environment, and leave out the rest.

POJO

A *Plain Old Java Object (POJO)* is a modular, reusable Java object. The name is used to emphasize that a given object is an ordinary Java Object, not a special object, and in particular not an Enterprise JavaBean. The term was coined by Martin Fowler, Rebecca Parsons and Josh MacKenzie in September 2000 in a talk where they were pointing out the many benefits of encoding business logic into regular java objects rather than using Entity Beans.

Java Bean

A Java Bean is a reusable software component that can be manipulated visually in a builder tool.

A Java Bean is an independent piece of code. It is not required to inherit from any particular base class or interface. Although Java Beans are primarily created in graphical IDEs, they can also be developed in simple text editors.

AOP

Aspect-Oriented Programming (AOP) is a programming paradigm in which secondary or supporting functions are isolated from the main program's business logic. It is a subset of object-oriented programming.

2.3. INSTALLATION

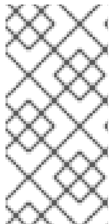
The Microcontainer is an integral part of the Enterprise Platform. More information about installing and configuring the Enterprise Platform can be found in the Administration and configuration Guide.

CHAPTER 3. BUILDING SERVICES

Services are pieces of code which perform work needed by multiple clients. For our purposes, we will put some additional constraints on the definition of a service. Services should have unique names which can be referenced, or called, by clients. The internals of a service should be invisible and unimportant to clients. This is the "black box" concept of object-oriented programming (OOP). In OOP, each object is independent, and no other object needs to know how it does its job.

In the context of the Microcontainer, services are built from POJOs. A POJO is nearly a service in its own right, but it can't be accessed by a unique name, and it must be created by the client that needs it.

Although a POJO must be created at run-time by the client, it does not need to be implemented by a separate class in order to provide a well-defined interface. As long as fields and methods are not removed, and access to them is not restricted, there is no need to recompile clients to use a newly-created POJO.



NOTE

Implementing an interface is only necessary in order to allow a client to choose between *alternative implementations*. If the client is compiled against an interface, many different implementations of the interface can be provided without having to recompile the client. The interface ensures that the method signatures do not change.

The remainder of this guide consists of creating a Human Resources service, using the Microcontainer to capture and modularize the business logic of the application. After the Microcontainer is installed, the example code is located in **`examples/User_Guide/gettingStarted/humanResourcesService`**.

3.1. INTRODUCTION TO THE HUMAN RESOURCES EXAMPLE

As you familiarize yourself with the directory structure of the files in the example, note that it uses the Maven Standard Directory Layout.

The Java source files are located in packages beneath the **`examples/User_Guide/gettingStarted/humanResourcesService/src/main/java/org/jboss/example/service`** directory, after you have extracted the ZIP file. Each of these classes represents a simple POJO that doesn't implement any special interfaces. The most important class is **`HRManager`**, which represents the service entry point providing all of the public methods that clients will call.

Methods Provided by the `HRManager` Class

- **`addEmployee`**(Employee ***`employee`***)
- **`removeEmployee`**(Employee ***`employee`***)
- **`getEmployee`**(String ***`firstName`***, String ***`lastName`***)
- **`getEmployees`**()
- **`getSalary`**(Employee ***`employee`***)
- **`setSalary`**(Employee ***`employee`***, Integer ***`newSalary`***)
- **`isHiringFreeze`**()

- **setHiringFreeze**(boolean ***hiringFreeze***)
- **getSalaryStrategy**()
- **setSalaryStrategy**(SalaryStrategy ***strategy***)

The Human Resources Service is composed of a handful of classes which maintain a list of employees and their details (addresses and salaries, in this case). Using the **SalaryStrategy** interface it is possible to configure the HRManager so that different salary strategy implementations are available to place minimum and maximum limits on the salaries for different employee roles.

3.2. COMPILING THE HRMANAGER EXAMPLE PROJECT

To compile the source code, issue **mvn compile** from the **humanResourcesService/** directory. This creates a new directory called **target/classes** which contains the compiled classes. To clean up the project and remove the target directory, issue the **mvn clean** command.

3.3. CREATING POJOS

Before a POJO can be used, you need to create it. You need a naming mechanism that allows you to register a reference to the POJO instance with a name. Clients need this name to use the POJO.

The Microcontainer provides such a mechanism: a *Controller*. A Controller allows you to deploy your POJO-based services into a run-time environment.

3.3.1. XML Deployment Descriptors

After compiling the classes, use an XML deployment descriptor to create instances of them. The descriptor contains a list of beans representing individual instances. Each bean has a unique name, so that it can be called by clients at run-time. The following descriptor deploys an instance of the HRManager:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">
  <bean name="HRService" class="org.jboss.example.service.HRManager"/>
</deployment>
```

This XML creates an instance of the **HRManager** class and registers it with the name HRService. This file is passed to an XML deployer associated with the Microcontainer at run-time, which performs the actual deployment, and instantiates the beans.

3.4. CONNECTING POJOS TOGETHER

Individual POJO instances can only provide relatively simple behavior. The true power of POJOs comes from connecting them together to perform complex tasks. How can you wire POJOs together to choose different salary strategy implementations?

The following XML deployment descriptor does just that:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">
  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <property name="salaryStrategy"><inject
bean="AgeBasedSalary"/></property>
  </bean>
  <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy"/>
</deployment>
```

This XML creates an instance of the chosen salary strategy implementation by including an additional <bean> element. This time, the AgeBasedSalaryStrategy is chosen. Next the code injects a reference to this bean into the instance of **`HRManager`** created using the HRService bean. Injection is possible because the **`HRManager`** class contains a **`setSalaryStrategy(SalaryStrategy strategy)`** method. Behind the scenes, JBoss Microcontainer calls this method on the newly created HRManager instance and passes a reference to the **`AgeBasedSalaryStrategy`** instance.

The XML deployment descriptor causes the same sequence of events to occur as if you had written the following code:

```
HRManager hrService = new HRManager();
AgeBasedSalaryStrategy ageBasedSalary = new AgeBasedSalaryStrategy();
hrService.setSalaryStrategy(ageBasedSalary);
```

In addition to performing injection via property setter methods, JBoss Microcontainer can also perform injection via constructor parameters if necessary. For more details please see the 'Injection' chapter in Part II 'POJO Development.'

3.4.1. Special Considerations

Although creating instances of classes using the <bean> element in the deployment descriptor is possible, it is not always the best way. For example, creating instances of the **`Employee`** and **`Address`** classes is unnecessary, because the client creates these in response to input from the user. They remain part of the service but are not referenced in the deployment descriptor.

Comment Your Code

You can define multiple beans within a deployment descriptor as long as each has a unique name, which is used to perform injection as shown above. However all of the beans do not necessarily represent services. While a service can be implemented using a single bean, multiple beans are usually used together. One bean typically represents the service entry point, and contains the public methods called by the clients. In this example the entry point is the HRService bean. The XML deployment descriptor does not indicate whether a bean represents a service or whether a bean is the service entry point. It is a good idea to use comments and an obvious naming scheme to delineate service beans from non-service beans.

3.5. WORKING WITH SERVICES

After creating POJOs and connecting them together to form services, you need to configure the services, test them, and package them.

3.5.1. Configuring A Service

Services can be configured by at least two ways:

- Injecting references between POJO instances
- Injecting values into POJO properties

In this example, the second method is used. The following deployment descriptor configures the HRManager instance in the following ways:

- A hiring freeze is implemented.
- The **AgeBasedSalaryStrategy** implements new minimum and maximum salary values.

Injecting references between POJO instances is one way of configuring a service however we can also inject values into POJO properties. The following deployment descriptor shows how we can configure the HRManager instance to have a hiring freeze and the AgeBasedSalaryStrategy to have new minimum and maximum salary values:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-deployer_2_0.xsd"
xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <property name="hiringFreeze">false</property>
    <property name="salaryStrategy"><inject
bean="AgeBasedSalary"/></property>
  </bean>

  <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
    <property name="minSalary">1000</property> <property
name="maxSalary">80000</property>
  </bean>

</deployment>
```

The classes must have public setter methods for the relevant properties so that values can be injected. For example, the **HRManager** class has a **setHiringFreeze(boolean hiringFreeze)** method and the **AgeBasedSalaryStrategy** class has **setMinSalary(int minSalary)** and **setMaxSalary(int maxSalary)** methods.

The values in the deployment descriptor are converted from strings into the relevant types (boolean, int etc...) by JavaBean PropertyEditors. Many PropertyEditors are provided by default for standard types, but you can create your own if necessary. See the Properties chapter in Part II 'POJO Development' for more details.

3.5.2. Testing A Service

After you have created your POJOs and connected them together to form services, you need to test them. JBoss Microcontainer allows unit testing individual POJOs as well as services, through the use of a **MicrocontainerTest** class.

The **org.jboss.test.kernel.junit.MicrocontainerTest** class inherits from **junit.framework.TestCase**, setting up each test by bootstrapping JBoss Microcontainer and adding

a BasicXMLDeployer. It then searches the classpath for an XML deployment descriptor with the same name as the test class, ending in **.xml** and residing in a directory structure representing the class's package name. Any beans found in this file are deployed and can then be accessed using a convenience method called **getBean(String name)**.

Examples of these deployment descriptors can be found in the Example 3.1, “Listing of the src/test/resources Directory”.

Example 3.1. Listing of the src/test/resources Directory

```
├── log4j.properties
└── org
    └── jboss
        └── example
            └── service
                ├── HRManagerAgeBasedTestCase.xml
                ├── HRManagerLocationBasedTestCase.xml
                ├── HRManagerTestCase.xml
                └── util
                    ├── AgeBasedSalaryTestCase.xml
                    └── LocationBasedSalaryTestCase.xml
```

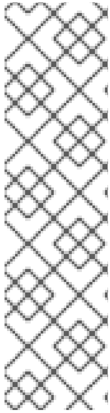
The test code is located in the src/test/java directory:

Example 3.2. Listing of the src/test/java Directory

```
└── org
    └── jboss
        └── example
            └── service
                ├── HRManagerAgeBasedTestCase.java
                ├── HRManagerLocationBasedTestCase.java
                ├── HRManagerTestCase.java
                ├── HRManagerTest.java
                ├── HRManagerTestSuite.java
                └── util
                    ├── AgeBasedSalaryTestCase.java
                    ├── LocationBasedSalaryTestCase.java
                    └── SalaryStrategyTestSuite.java
```

The **HRManagerTest** class extends **MicrocontainerTest** in order to set up a number of employees to use as the basis for the tests. Individual test cases then subclass **HRManagerTest** to perform the actual work. Also included are a couple of **TestSuite** classes that are used to group individual test cases together for convenience.

To run the tests, enter **mvn test** from the **humanResourcesService/** directory. You should see some **DEBUG** log output which shows JBoss Microcontainer starting up and deploying beans from the relevant XML file before running each test. At the end of the test the beans are undeployed and the Microcontainer is shut down.



NOTE

Some of the tests, such as **HRManagerTestCase**, **AgeBasedSalaryTestCase**, and **LocationBasedSalaryTestCase**, unit test individual POJOs. Other tests, such as **HRManagerAgeBasedTestCase** and **HRManagerLocationBasedTestCase** unit test entire services. Either way, the tests are run in the same manner. Using the MicrocontainerTest class makes it easy to set up and conduct comprehensive tests for any part of your code.

The **Address** and **Employee** classes are not tested here. Writing tests for them is up to you.

3.5.3. Packaging A Service

After testing your service, it is time to package it up so that others can use it. The simplest way to do this is to create a JAR containing all of the classes. You can choose to include the deployment descriptor if there is a sensible default way to configure the service, but this is optional.

Procedure 3.1. Packaging a Service

1. **Place the deployment descriptor in the META-INF directory (optional)**
   If you do choose to include the deployment descriptor, by convention it should be named **jboss-beans.xml** and should be placed in a **META-INF** directory. This is the default layout for the Enterprise Platform, so the JAR deployer recognizes this layout and automatically performs the deployment.

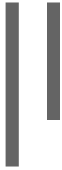
   The deployment descriptor is not included in the Human Resources example, because the service is configured by editing the descriptor directly, as a separate file.

2. **Generate the JAR**
   To generate a JAR containing all of the compiled classes, enter **mvn package** from the **humanResourcesService/** directory.

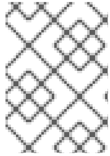
3. **Make the JAR available to other Maven projects**
   To make the JAR available to other Maven projects, enter **mvn install** in order to copy it to your local Maven repository. The final layout of the JAR is shown in Example 3.3, “Listing of the **org/jboss/example/service** and **META-INF**Directories”.

Example 3.3. Listing of the org/jboss/example/service and META-INFDirectories

```
`-- org
    `-- jboss
        `-- example
            `-- service
                |-- Address.java
                |-- Employee.java
                |-- HRManager.java
                `-- util
                    |-- AgeBasedSalaryStrategy.java
                    |-- LocationBasedSalaryStrategy.java
                    `-- SalaryStrategy.java
`--META-INF
    `-- MANIFEST.MF
```



```
`-- maven
    `-- org.jboss.micrcontainer.examples
     `-- humanResourceService
```



NOTE

The **META-INF/maven** directory is automatically created by Maven, and will not be present if you are using a different build system.

CHAPTER 4. USING SERVICES

The previous chapter guided you through creating, configuring, testing and packaging a service. The next step is to create a client which will perform actual work using the service.

The client in this example uses a *Text User Interface (TUI)* to accept input from the user and output results. This reduces the size and complexity of the example code.

All of the necessary files are located in the **`examples/User_Guide/gettingstarted/commandLineClient`** directory, which follows the Maven Standard Directory Layout, as seen in Example 4.1, "Listing for **`examples/User_Guide/gettingstarted/commandLineClient`** Directory".

Example 4.1. Listing for `examples/User_Guide/gettingstarted/commandLineClient` Directory

```
├── pom.xml
├── src
│   ├── main
│   │   ├── assembly
│   │   │   ├── aop.xml
│   │   │   ├── classloader.xml
│   │   │   ├── common.xml
│   │   │   └── pojo.xml
│   │   ├── config
│   │   │   ├── aop-beans.xml
│   │   │   ├── classloader-beans.xml
│   │   │   ├── pojo-beans.xml
│   │   │   └── run.sh
│   │   ├── java
│   │   │   └── org
│   │   │       └── jboss
│   │   │           └── example
│   │   │               └── client
│   │   │                   ├── Client.java
│   │   │                   ├── ConsoleInput.java
│   │   │                   ├── EmbeddedBootstrap.java
│   │   │                   └── UserInterface.java
│   │   └── resources
│   │       └── log4j.properties
│   └── test
│       ├── java
│       │   └── org
│       │       └── jboss
│       │           └── example
│       │               └── client
│       │                   ├── ClientTestCase.java
│       │                   ├── ClientTestSuite.java
│       │                   └── MockUserInterface.java
│       └── resources
│           └── jboss-beans.xml
└── target
    └── classes
        └── log4j.properties
```

The client consists of three classes and one interface, located in the **org/jboss/example/client** directory.

UserInterface describes methods that the client calls at run-time to request user input. **ConsoleInput** is an implementation of **UserInterface** that creates a TUI which the user uses to interact with the client. The advantage of this design is that you can easily create a Swing implementation of **UserInterface** at a later date and replace the TUI with a GUI. You can also simulate the data-entry process with a script. Then you can check the behavior of the client automatically using conventional JUnit test cases found in Example 3.2, “Listing of the **src/test/java** Directory”.

For the build to work you must first build and install **auditAspect.jar** from the **examples/User_Guide/gettingStarted/auditAspect** directory using the **mvn install command**. A number of different client distributions are created, including one based on AOP which relies on **auditAspect.jar** being available in the local Maven repository.

If you previously typed **mvn install** from the **examples/User_Guide/gettingStarted** directory then **humanResourcesService.jar** and **auditAspect.jar** have already been built and packaged, along with the client, so this step will not be necessary.

To compile the source code, all of the steps in Procedure 4.1, “Compiling the Source Code” are performed when you issue the **mvn package** command from the **commandLineClient** directory.

Procedure 4.1. Compiling the Source Code

1. Run the unit tests.

2. Build a client JAR.

3. Assemble a distribution containing all of the necessary files.

After compiling and packaging the client, the directory structure in the **commandLineClient/target** directory includes the subdirectories described in Example 4.2, “Subdirectories of the **commandLineClient/target** Directory”.

Example 4.2. Subdirectories of the commandLineClient/target Directory

client-pojo

used to call the service without AOP.

client-cl

used to demonstrate classloading features.

client-aop

Adding AOP support. See Chapter 5, *Adding Behavior with AOP* for more details.

Each sub-directory represents a different distribution with all of the shell scripts, JARs, and XML deployment descriptors needed to run the client in different configurations. The rest of this chapter uses the **client-pojo** distribution found in the **client-pojo** sub-directory, which is listed in Example 4.3, “Listing of the **client-pojo** Directory”.

Example 4.3. Listing of the client-pojo Directory

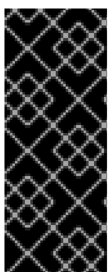
```
        |-- client-1.0.0.jar
        |-- jboss-beans.xml
        |-- lib
        |   |-- concurrent-1.3.4.jar
        |   |-- humanResourcesService-1.0.0.jar
        |   |-- jboss-common-core-2.0.4.GA.jar
        |   |-- jboss-common-core-2.2.1.GA.jar
        |   |-- jboss-common-logging-log4j-2.0.4.GA.jar
        |   |-- jboss-common-logging-spi-2.0.4.GA.jar
        |   |-- jboss-container-2.0.0.Beta6.jar
        |   |-- jboss-dependency-2.0.0.Beta6.jar
        |   |-- jboss-kernel-2.0.0.Beta6.jar
        |   |-- jbossxb-2.0.0.CR4.jar
        |   |-- log4j-1.2.14.jar
        |   `-- xercesImpl-2.7.1.jar
        `-- run.sh
```

To run the client, change to the **client-pojo** directory and type **./run.sh**. The Example 4.4, “HRManager Menu Screen” appears.

Example 4.4. HRManager Menu Screen

```
Menu:

d) Deploy Human Resources service
u) Undeploy Human Resources service

a) Add employee
l) List employees
r) Remove employee
g) Get a salary
s) Set a salary
t) Toggle hiring freeze

m) Display menu
p) Print service status
q) Quit
>
```

To select an option, enter the letter shown on the left-hand side and press **RETURN**. For example to display the menu options enter **m** followed by **RETURN**. Entering more than one letter or entering an invalid option results in an error message.



IMPORTANT

The **run.sh** script sets up the run-time environment by adding all of the JARs found in the **lib/** directory to the classpath using the java.ext.dirs system property. It also adds the current directory and the **client-1.0.0.jar** using the -cp flag so that the **jboss-beans.xml** deployment descriptor can be found at run-time along with the **org.jboss.example.client.Client** class which is called to start the application.

4.1. BOOTSTRAPPING THE MICROCONTAINER

Before using the client to deploy and call your service, take a closer look at what happened during its construction:

```
public Client(final boolean useBus) throws Exception {
    this.useBus = useBus;

    ClassLoader cl = Thread.currentThread().getContextClassLoader();
    url = cl.getResource("jboss-beans.xml");

    // Start JBoss Microcontainer
    bootstrap = new EmbeddedBootstrap();
    bootstrap.run();

    kernel = bootstrap.getKernel();
    controller = kernel.getController();
    bus = kernel.getBus();
}
```

First of all a URL representing the **`jboss-beans.xml`** deployment descriptor is created. This is later required so that the XML deployer can deploy and undeploy beans declared in the file. The **`getResource()`** method of the application classloader is used because the **`jboss-beans.xml`** file is included on the classpath. This is optional; the name and location of the deployment descriptor are unimportant as long as the URL is valid and reachable.

Next an instance of JBoss Microcontainer is created, along with an XML deployer. This process is called *bootstrapping* and a convenience class called **`BasicBootstrap`** is provided as part of the Microcontainer to allow for programmatic configuration. To add an XML deployer, extend **`BasicBootstrap`** to create an **`EmbeddedBootstrap`** class and override the protected **`bootstrap()`** method as follows:

```
public class EmbeddedBootstrap extends BasicBootstrap {
    protected BasicXMLDeployer deployer;
    public EmbeddedBootstrap() throws Exception {
 super();
    }

    public void bootstrap() throws Throwable {
 super.bootstrap();
 deployer = new BasicXMLDeployer(getKernel());
 Runtime.getRuntime().addShutdownHook(new Shutdown());
    }

    public void deploy(URL url) {
 ...
 deployer.deploy(url);
 ...
     }
```

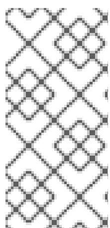
```
    public void undeploy(URL url) {
 ...
 deployer.undeploy(url);
 ...
     }

    protected class Shutdown extends Thread {
 public void run() {
     log.info("Shutting down");
     deployer.shutdown();
 }
    }
}
```

The **shutdown** hook ensures that when the JVM exits, all of the beans are undeployed in the correct order. The public **deploy**/**undeploy** methods delegate to the **BasicXMLDeployer** so that beans declared in **jboss-beans.xml** can be deployed and undeployed.

Finally references to the Microcontainer controller and bus are restored, so you can look up bean references by name and access them directly or indirectly as necessary.

4.2. DEPLOYING THE SERVICE

After creating the client, you can deploy the Human Resources service. This is done by entering the **d** option from the TUI. Output indicates that the **BasicXMLDeployer** has parsed the **jboss-beans.xml** file using the URL, and instantiated the beans found within.



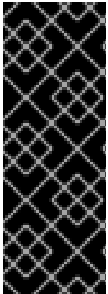
NOTE

The Microcontainer is able to instantiate the beans because their classes are available in the extension classpath inside the **lib/humanResourcesService.jar** file. You can also place these classes in an exploded directory structure and add it to the application classpath, but packaging them in a JAR is typically more convenient.

The deployment descriptor is entirely separate from the **humanResourcesService.jar** file. This enables easy editing of it for testing purposes. The **jboss-beans.xml** file in the example contains some commented-out fragments of XML which show some of the possible configurations.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <!-- <property name="hiringFreeze">true</property>
  <property name="salaryStrategy"><inject bean="AgeBasedSalary"/>
</property> -->
```

```
  </bean>

  <!-- <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
       <property name="minSalary">1000</property>
       <property name="maxSalary">80000</property>
     </bean>

     <bean name="LocationBasedSalary"
    class="org.jboss.example.service.util.LocationBasedSalaryStrategy">
       <property name="minSalary">2000</property>
       <property name="maxSalary">90000</property>
       </bean> -->

</deployment>
```



IMPORTANT

Depending on how you access the service at run-time, you may need to shut down the application and restart it again to redeploy the service and see your changes. This reduces the flexibility of the application, but results in faster performance at run-time. Alternatively you may be able to simply redeploy the service while the application is running. This increases flexibility but results in slower run-time performance. Keep these trade-offs under consideration when designing your applications.

4.3. DIRECT ACCESS

If no parameters are given to the **run.sh** script when the client is started, a reference to the **HRService** bean is looked up using the Microcontainer controller after the service is deployed:

```
private HRManager manager;
...
private final static String HRSERVICE = "HRService";

...
void deploy() {
    bootstrap.deploy(url);
    if (!useBus && manager == null) {
 ControllerContext context = controller.getInstalledContext(HRSERVICE);
 if (context != null) { manager = (HRManager) context.getTarget(); }
    }
}
```

Rather than immediately looking up a reference to the bean instance, the example first looks up a

reference to a **ControllerContext**, then obtains a reference to the bean instance from the context using the **getTarget()** method. The bean can exist within the Microcontainer in any of the states listed in States of a Bean Within the Microcontainer.

States of a Bean Within the Microcontainer

- NOT_INSTALLED
- DESCRIBED
- INSTANTIATED
- CONFIGURED
- INSTALLED

To keep track of which state the bean is in, wrap it in another object called a *context* that describes the current state. The name of the context is the same as the bean name. Once a context reaches the INSTALLED state, the bean it represents is considered to be deployed.

After creating a reference to the bean instance representing the service entry point, you can call methods on it to preform work:

```
@SuppressWarnings("unchecked")

    Set<Employee> listEmployees() {
    if (useBus)
 ...
    else
 return manager.getEmployees();
}
```

The client is accessing the service directly since it is using a reference to the actual bean instance. Performance is good, because each method call goes directly to the bean. What happens, however, if you want to reconfigure the service and redeploy it while the application is running?

Reconfiguration is achieved by making changes to the XML deployment descriptor and saving the file. In order to redeploy the service, the current instance must be undeployed. During undeployment the Microcontainer controller releases its reference to the bean instance, along with any dependent beans. These beans will subsequently become available for garbage collection because they are no longer required by the application. Redeploying the service creates new bean instances representing the new configuration. Any subsequent look-ups from clients will retrieve references to these new instances and they will be able to access the reconfigured service.

The problem is that the reference to the bean instance representing our service entry point is cached when you deploy the service for the first time. Undeploying the service has no affect, since the bean instance can still be accessed using the cached reference and it will not be garbage collected until the client releases it. Along the same line, deploying the service again will not cause another look-up because the client already has a cached reference. It will therefore continue to use the bean instance representing the initial service configuration.

You can test this behavior by typing **u** followed by **RETURN** to undeploy the current service. You should still be able to access the service from the client even though it is 'undeployed'. Next, make some changes to the configuration using the **jboss-beans.xml** file, save the file, and deploy it again using the **d** option. Printing out the status of the service using the**p** option shows that the client is still accessing the initial instance of the service that was deployed.

WARNING

Even if you modify the client to look up a new reference each time the service is redeployed, new developers may hand out copies of this reference to other objects, by mistake. If all of these references are not cleaned up during redeployment, the same caching problem can occur.

To reliably redeploy the reconfigured service, shut down the application completely using the 'q' option and restart it again using the **run.sh** script. For enterprise services such as Transactions, Messaging and Persistence this is perfectly acceptable behavior, since they are generally always in use. They cannot be redeployed at run-time and also benefit from the high performance given by using direct access. If your service falls into this category, consider using direct access via the Microcontainer controller.

4.4. INDIRECT ACCESS

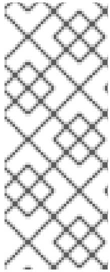
The **run.sh** script can be called with an optional parameter bus, which causes calls to the Human Resources service to use the Microcontainer bus.

Instead of using a direct reference to the bean instance obtained from the Microcontainer controller, the new behavior is to call an **invoke()** method on the bus, passing in the bean name, method name, method arguments and method types. The bus uses this information to call the bean on the client's behalf.

```
private final static String HRSERVICE = "HRService";

...

    @SuppressWarnings("unchecked")
 Set<Employee> listEmployees() {
 if (useBus)
     return (Set<Employee>) invoke(HRSERVICE, "getEmployees", new Object[]
{}, new String[] {});
 else
     return manager.getEmployees();
    }

private Object invoke(String serviceName, String methodName, Object[]
args, String[] types) {
    Object result = null;
    try {
 result = bus.invoke(serviceName, methodName, args, types);
    } catch (Throwable t) {
 t.printStackTrace();
```

```
        }
        return result;
    }
```

The bus looks up the reference to the named bean instance and calls the chosen method using reflection. The client never has a direct reference to the bean instance, so it is said to accesses the service indirectly. Since the bus does not cache the reference, you can safely make changes to the service configuration, and it can be redeployed at run-time. Subsequent calls by the client will use the new reference, as expected. The client and service have been decoupled.



NOTE

This behavior can be tested by deploying the service and using the **p** option to print out the status. Undeploy the service using the **u** option and notice that it is inaccessible. Next, make some changes to **jboss-beans.xml** file, save the changes, and deploy it again using the **d** option. Print out the status again using the **p** option. The client is accessing the new service configuration.

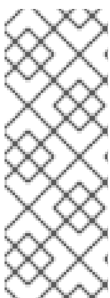
Because the bus uses reflection to call bean instances, it is a slower than direct access. The benefit of the approach is that only the bus has references to the bean instances. When a service is redeployed, all of the existing references can be cleaned up and replaced with new ones. This way, a service can be reliably redeployed at run-time. Services that are not used very often or that are specific to certain applications are good candidates for indirect access using the Microcontainer bus. Often, the reduction in performance is outweighed by the flexibility that this provides.

4.5. DYNAMIC CLASSLOADING

So far you have used the extension and application classloaders to load all of the classes in the application. The application classpath is set up by the **run.sh** script using the -cp flag to include the current directory and the **client-1.0.0.jar**, as shown here:

```
java -Djava.ext.dirs=`pwd`/lib -cp .:client-1.0.0.jar
org.jboss.example.client.Client $1
```

For convenience the JARs in the **lib** directory were added to the extension classloader's classpath using the java.ext.dirs system property, rather than listing the full path to each of the JARs after the -cp flag. Because the **classloader** extension is the parent of the **classloader** application, the client classes is able to find all of the Microcontainer classes and the Human Resources service classes at run-time.



NOTE

With Java versions 6 and higher, you can use a wild-card to include all JARs in a directory with the -cp flag: **java -cp `pwd`/lib/*:.:client-1.0.0.jar org.jboss.example.client.Client $1**

Here, all of the classes in the application will be added to the application classloader's classpath, and the extension classloader's classpath will retain its default value.

What happens if you need to deploy an additional service at run-time? If the new service is packaged in a JAR file, it must be visible to a classloader before any of its classes can be loaded. Because you have already set up the classpath for the application classloader (and extension classloader) on start-up, it is

not easy to add the URL of the JAR. The same situation applies if the service classes are contained in a directory structure. Unless the top-level directory is located in the current directory (which is on the application classpath) then the classes will not be found by the application classloader.

If you wish to redeploy an existing service, changing some of its classes, you need to work around security constraints, which forbid an existing classloader from reloading classes.

The goal is to create a new classloader that knows the location of the new service's classes, or that can load new versions of an existing service's classes, in order to deploy the service's beans. JBoss Microcontainer uses the <classloader> element in the deployment descriptor to accomplish this.

The **client-cl** distribution contains the file listed in the Example 4.5, “Listing of the **commandLineClient/target/client-cl** Directory” .

Example 4.5. Listing of the commandLineClient/target/client-cl Directory

```
|-- client-1.0.0.jar
|-- jboss-beans.xml
|-- lib
|   |-- concurrent-1.3.4.jar
|   |-- jboss-common-core-2.0.4.GA.jar
|   |-- jboss-common-core-2.2.1.GA.jar
|   |-- jboss-common-logging-log4j-2.0.4.GA.jar
|   |-- jboss-common-logging-spi-2.0.4.GA.jar
|   |-- jboss-container-2.0.0.Beta6.jar
|   |-- jboss-dependency-2.0.0.Beta6.jar
|   |-- jboss-kernel-2.0.0.Beta6.jar
|   |-- jbossxb-2.0.0.CR4.jar
|   |-- log4j-1.2.14.jar
|   `-- xercesImpl-2.7.1.jar
|-- otherLib
|   `-- humanResourcesService-1.0.0.jar
|`-- run.sh
```

The **humanResourcesService.jar** file has been moved to a new sub-directory called **otherLib**. It is no longer available to either the extension or application classloaders whose classpaths are setup in the run.sh script:

```
java -Djava.ext.dirs=`pwd`/lib -cp .:client-1.0.0.jar
org.jboss.example.client.Client $1
```

To work around this, create a new classloader during the deployment of the service, load it in the service classes, and create instances of the beans. To see how this is done, look at the contents of the **jboss-beans.xml** file:

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="URL" class="java.net.URL">
```

```
    <constructor>

<parameter>file:/Users/newtonm/jbossmc/microcontainer/trunk/docs/examples/
User_Guide/gettingStarted/commandLineClient/target/client-
cl.dir/otherLib/humanResourcesService-1.0.0.jar</parameter>
    </constructor>
  </bean>

  <bean name="customCL" class="java.net.URLClassLoader">
    <constructor>
      <parameter>
 <array>
   <inject bean="URL"/>
 </array>
      </parameter>
    </constructor>
  </bean>

  <bean name="HRService" class="org.jboss.example.service.HRManager">
    <classloader><inject bean="customCL"/></classloader>
    <!-- <property name="hiringFreeze">true</property>
  <property name="salaryStrategy"><inject bean="AgeBasedSalary"/>
</property> -->
  </bean>

  <!-- <bean name="AgeBasedSalary"
class="org.jboss.example.service.util.AgeBasedSalaryStrategy">
       <property name="minSalary">1000</property>
       <property name="maxSalary">80000</property>
       </bean>

<bean name="LocationBasedSalary"
class="org.jboss.example.service.util.LocationBasedSalaryStrategy">
<property name="minSalary">2000</property>
<property name="maxSalary">90000</property>
</bean> -->

</deployment>
```

1. First, create an instance of **java.net.URL** called **URL**, using parameter injection in the constructor to specify the location of the **humanResourcesService.jar** file on the local file-system.

2. Next, create an instance of a **URLClassLoader** by injecting the URL bean into the constructor as the only element in an array.

3. Include a <classloader> element in your **HRService** bean definition and inject the **customCL** bean. This specifies that the **HRManager** class needs to be loaded by the customCL classloader.

You need a way to decide which classloader to use for the other beans in the deployment. All beans in the deployment use the current thread's context classloader. In this case the thread that handles

deployment is the main thread of the application which has its context classloader set to the application classloader on start-up. If you wish, you can specify a different classloader for the entire deployment using a <classloader> element, as shown in Example 4.6, “Specifying a Different Classloader”.

Example 4.6. Specifying a Different Classloader

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
deployer_2_0.xsd"
     xmlns="urn:jboss:bean-deployer:2.0">

  <classloader><inject bean="customCL"/></classloader>

  <bean name="URL" class="java.net.URL">
    <constructor>

<parameter>file:/Users/newtonm/jbossmc/microcontainer/trunk/docs/example
s/User_Guide/gettingStarted/commandLineClient/target/client-
cl.dir/otherLib/humanResourcesService-1.0.0.jar</parameter>
    </constructor>
  </bean>

  <bean name="customCL" class="java.net.URLClassLoader">
    <constructor>
      <parameter>
 <array>
   <inject bean="URL"/>
 </array>
      </parameter>
    </constructor>
  </bean>

  ...

</deployment>
```

This would be necessary to allow for reconfiguration of the service by uncommenting the **`AgeBasedSalary`** or **`LocationBasedSalary`** beans. Classloaders specified at the bean level override the deployment level classloader. To override the deployment level classloader altogether, and use the default classloader for a bean, use the <null/> value as follows:

```
<bean name="HRService" class="org.jboss.example.service.HRManager">
  <classloader><null/></classloader>
</bean>
```

4.5.1. Problems With Classloaders Created with Deployment Descriptors

If you create a new classloader for your service using the deployment descriptor, you may not be able to access classes loaded by it from the application classloader. In the HRManager example, the client is no longer able to cache a direct reference to the bean instance when using the Microcontainer controller.

To see this behavior, start the client using the **`run.sh`** command, then try to deploy the service. A **`java.lang.NoClassDefFoundError`** exception is thrown and the application exits.

In this scenario, you must use the bus to access the service indirectly and provide access to any classes shared by the client in the application classpath. In this example, the affected classes are **`Address`**, **`Employee`**, and **`SalaryStrategy`**.

CHAPTER 5. ADDING BEHAVIOR WITH AOP

Object Oriented Programming (OOP) contains many useful techniques for software development including encapsulation, inheritance, and polymorphism. However, it does not solve the problem of addressing logic that is often repeated in many different classes. Examples of this include logging, security, and transactional logic which is traditionally hard-coded into each class. This type of logic is called a *cross-cutting concern*.

Aspect Oriented Programming (AOP) works to allow cross-cutting concerns to be applied to classes after they have been compiled. This keeps the source code free of logic which is not central to the main purpose of the class and streamlines maintenance. The method depends on the AOP implementation. Typically if a class implements an interface, each method call to an instance of the class first passes through a proxy. This proxy implements the same interface, adding the required behavior. Alternatively, if an interface is not used, then the java bytecode of the compiled class is modified: the original methods are renamed and replaced by methods that implement the cross-cutting logic. These new methods then call the original methods after the cross-cutting logic has been executed. Another method to achieve the same result is modifying the bytecode to create a subclass of the original class that overrides its methods. The overridden methods then execute the cross-cutting logic before calling the corresponding methods of the super class.

JBoss AOP is a framework for AOP. Using it, you can create cross-cutting concerns using conventional java classes and methods. In AOP terminology each concern is represented by an *aspect* that you implement using a simple POJO. Behavior is provided by methods within the aspect called *advices*. These advices follow certain rules for their parameter, and return types and any exceptions that they throw. Within this framework, you can use conventional object-oriented notions such as inheritance, encapsulation, and composition to make your cross-cutting concerns easy to maintain. Aspects are applied to code using an expression language that allows you to specify which constructors, methods and even fields to target. You can quickly change the behavior of multiple classes by editing a configuration file.

This chapter contains examples which demonstrate how to use JBoss AOP alongside the Microcontainer to create and apply an auditing aspect to the Human Resources Service. The auditing code could be placed within the **`HRManager`** class, but it would clutter the class with code which is not relevant to its central purpose, bloating it and making it harder to maintain. The design of the aspect also provide modularity, making it easy to audit other classes in the future, if the scope of the project changes.

AOP can also be used to apply additional behavior during the deployment phase. This example will create and bind a proxy to a bean instance into a basic JNDI service, allowing it to be accessed using a JNDI look-up instead of the Microcontainer controller.

5.1. CREATING AN ASPECT

The **`examples/User_Guide/gettingStarted/auditAspect`** directory contains all the files needed to create the aspect.

- **`pom.xml`**
- **`src/main/java/org/jboss/example/aspect/AuditAspect.java`**

Example 5.1. Example POJO

```
public class AuditAspect {

    private String logDir;
    private BufferedWriter out;
```

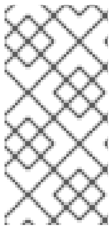
```
    public AuditAspect() {
        logDir = System.getProperty("user.dir") + "/log";

        File directory = new File(logDir);
        if (!directory.exists()) {
            directory.mkdir();
        }
    }

    public Object audit(ConstructorInvocation inv) throws Throwable {
        SimpleDateFormat formatter = new SimpleDateFormat("ddMMyyyy-
kkmmss");
        Calendar now = Calendar.getInstance();
        String filename = "auditLog-" +
formatter.format(now.getTime());

        File auditLog = new File(logDir + "/" + filename);
        auditLog.createNewFile();
        out = new BufferedWriter(new FileWriter(auditLog));
        return inv.invokeNext();
    }

    public Object audit(MethodInvocation inv) throws Throwable {
        String name = inv.getMethod().getName();
        Object[] args = inv.getArguments();
        Object retVal = inv.invokeNext();

        StringBuffer buffer = new StringBuffer();
        for (int i=0; i < args.length; i++) {
            if (i > 0) {
                buffer.append(", ");
            }
            buffer.append(args[i].toString());
        }

        if (out != null) {
            out.write("Method: " + name);
            if (buffer.length() > 0) {
                out.write(" Args: " + buffer.toString());
            }
            if (retVal != null) {
  out.write(" Return: " + retVal.toString());
            }
            out.write("\n");
            out.flush();
        }
        return retVal;
    }
}
```

Procedure 5.1. Creating the POJO

1. The constructor checks for the presence of a **log** directory in the current working directory, and creates one if not found.

2. Next, an advice is defined. This advice is called whenever the constructor of the target class is called. This creates a new log file within the **log** directory to record method calls made on different instances of the target class in separate files.

3. Finally, another advice is defined. This advice applies to each method call made on the target class.The method name and arguments are stored, along with the return value. This information is used to construct an audit record and write it to the current log file. Each advice calls **inv.invokeNext()**, which chains the advices together if more than one cross-cutting concern has been applied, or to call the target constructor/method.



NOTE

Each advice is implemented using a method that takes an invocation object as a parameter, throws Throwable and returns Object. At design time you don't know which constructors or methods these advices will be applied to, so make the types as generic as possible.

To compile the class and create an **auditAspect.jar** file that can be used by other examples, type **mvn install** from the **auditAspect** directory.

5.2. CONFIGURING THE MICROCONTAINER FOR AOP

Before applying the audit aspect to the HR Service, a number of JARs must be added to the extension classpath. They are in the **lib** sub-directory of the **client-aop** distribution located in the **examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir** directory:

Example 5.2. Listing of the examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir Directory

```
|-- client-1.0.0.jar
|-- jboss-beans.xml
|-- lib
|-- auditAspect-1.0.0.jar
|-- concurrent-1.3.4.jar
|-- humanResourcesService-1.0.0.jar
|-- javassist-3.6.0.GA.jar
|-- jboss-aop-2.0.0.beta1.jar
|-- jboss-aop-mc-int-2.0.0.Beta6.jar
|-- jboss-common-core-2.0.4.GA.jar
|-- jboss-common-core-2.2.1.GA.jar
|-- jboss-common-logging-log4j-2.0.4.GA.jar
|-- jboss-common-logging-spi-2.0.4.GA.jar
|-- jboss-container-2.0.0.Beta6.jar
|-- jboss-dependency-2.0.0.Beta6.jar
|-- jboss-kernel-2.0.0.Beta6.jar
|-- jbossxb-2.0.0.CR4.jar
|-- log4j-1.2.14.jar
|-- trove-2.1.1.jar
```

```
        `-- xercesImpl-2.7.1.jar
        |-- log
        `-- auditLog-18062010-122537
        `-- run.sh
```

First, **lib/auditAspect-1.0.0.jar** is required to create an instance of the aspect at run-time, in order to execute the logic. Next the jar file for JBoss AOP (jboss-aop.jar), along with its dependencies javassist and trove, adds the AOP functionality. Finally, the jboss-aop-mc-int jar is required because it contains an XML schema definition that allows you to define aspects inside an XML deployment descriptor. It also contains integration code to create dependencies between normal beans and aspect beans within the Microcontainer, allowing you to add behavior during the deployment and undeployment phases.

Because you are using Maven2 to assemble the client-aop distribution, you should add these JAR files by declaring the appropriate dependencies in your **pom.xml** file and creating a valid assembly descriptor. A sample **pom.xml** snippet is shown in Example 5.3, “Example **pom.xml** Excerpt for AOP”. To perform your build using Ant, the procedure will be different.

Example 5.3. Example pom.xml Excerpt for AOP

```
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>jboss-oap</artifactId>
  <version>2.0.0</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>javassist</artifactId>
  <version>3.6.0.GA</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>trove</artifactId>
  <version>2.1.1</version>
</dependency>
<dependency>
  <groupId>org.jboss.microcontainer.examples</groupId>
  <artifactId>jboss-aop-mc-int</artifactId>
  <version>2.0.0.Beta6</version>
</dependency>
```

5.3. APPLYING AN ASPECT

Now that you have a valid distribution containing everything you need, you can configure **jboss-beans.xml** to apply the audit aspect. It is in **examples/User_Guide/gettingStarted/commandLineClient/target/client-aop.dir**.

```
<?xml version="1.0" encoding="UTF-8"?>

<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="urn:jboss:bean-deployer:2.0 bean-
```

```
deployer_2_0.xsd"
            xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="AspectManager" class="org.jboss.aop.AspectManager">
    <constructor factoryClass="org.jboss.aop.AspectManager"
   factoryMethod="instance"/>
  </bean>

  <aop:aspect xmlns:aop="urn:jboss:aop-beans:1.0"
       name="AuditAspect" class="org.jboss.example.aspect.AuditAspect"
       method="audit" pointcut="execution(public
org.jboss.example.service.HRManager->new(..)) OR
execution(public * org.jboss.example.service.HRManager->*(..))">
  </aop:aspect>

  ...

</deployment>
```

Procedure 5.2. Explanation of the Code to Apply an Aspect

1. Before you can apply your aspect to any classes, you need to create an instance of **`org.jboss.aop.AspectManager`** using a <bean> element. A factory method is used here instead of calling a conventional constructor, since only one instance of the AspectManager in the JVM is necessary at run-time.

2. Next an instance of our aspect called AuditAspect is created, using the <aop:aspect> element. This looks similar to the <bean> element because it has name and class attributes that are used in the same way. However it also has method and pointcut attributes that you can use to apply or bind an advice within the aspect to constructors and methods within other classes. These attributes bind the audit advice to all public constructors and methods within the **`HRManager`** class. Only the **`audit`** method needs to be specified, since it has been overloaded within the **`AuditAspect`** class with different parameters. JBoss AOP knows at run-time which to select, depending on whether a constructor or method invocation is being made.

This additional configuration is all that is needed to apply the audit aspect at run-time, adding auditing behavior to the **Human Resources** service. You can test this by running the client using the **`run.sh`** script. A **`log`** directory is created on start-up alongside the **`lib`** directory when the **`AuditAspect`** bean is created by the Microcontainer. Each deployment of the Human Resources service causes a new log file to appear within the **`log`** directory. The log file contains a record of any calls made from the client to the service. It is named something similar to **`auditLog-28112007-163902`**, and contains output similar to Example 5.4, “Example AOP Log Output”.

Example 5.4. Example AOP Log Output

```
        Method: getEmployees Return: []
        Method: addEmployee Args: (Santa Claus, 1 Reindeer Avenue,
Lapland City - 25/12/1860) Return: true
        Method: getSalary Args: (Santa Claus, null - Birth date
```

```
unknown) Return: 10000
        Method: getEmployees Return: [(Santa Claus, 1 Reindeer Avenue,
Lapland City - 25/12/1860)]
        Method: isHiringFreeze Return: false
        Method: getEmployees Return: [(Santa Claus, 1 Reindeer Avenue,
Lapland City - 25/12/1860)]
        Method: getSalaryStrategy
```

To remove the auditing behavior, comment out the relevant fragments of XML in the deployment descriptor and restart the application.

WARNING

The order of deployment matters. Specifically each aspect must be declared before the beans that it applies to, so that the Microcontainer deploys them in that order. This is because the Microcontainer may need to alter the bytecode of the normal bean class to add the cross-cutting logic, before it creates an instance and stores a reference to it in the controller. If a normal bean instance has already been created, this is not possible.

5.4. LIFECYCLE CALLBACKS

In addition to applying aspects to beans that we instantiate using the Microcontainer we can also add behavior during the deployment and undeployment process. As mentinoed in Section 4.3, “Direct Access”, a bean goes through several different states as it is deployed. These include:

NOT_INSTALLED

the deployment descriptor containing the bean has been parsed, along with any annotations on the bean itself.

DESCRIBED

any dependencies created by AOP have been added to the bean, and custom annotations have been processed.

INSTANTIATED

an instance of the bean has been created.

CONFIGURED

properties have been injected into the bean, along with any references to other beans.

CREATE

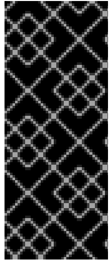
the **`create`** method, if defined in the bean, has been called.

START

the **`start`** method, if defined in the bean, has been called.

INSTALLED

any custom install actions that were defined in the deployment descriptor have been executed and the bean is ready to access.



IMPORTANT

The **CREATE** and **START** states are included for legacy purposes. This allows services that were implemented as MBeans in previous versions of the Enterprise Platform to function correctly when implemented as beans in the Enterprise Platform 5.1. If you do not define any corresponding create/start methods in your bean, it will pass straight through these states.

These states represent the bean's lifecycle. You can define a number of callbacks to be applied to any point by using an additional set of <aop> elements:

<aop:lifecycle-describe>

applied when entering/leaving the DESCRIBED state

<aop:lifecycle-instantiate>

applied when entering/leaving the INSTANTIATED state

<aop:lifecycle-configure>

applied when entering/leaving the CONFIGURED state

<aop:lifecycle-create>

applied when entering/leaving the CREATE state

<aop:lifecycle-start>

applied when entering/leaving the START state

<aop:lifecycle-install>

applied when entering/leaving the INSTALLED state

Like the <bean> and <aop:aspect> elements, the <aop:lifecycle-> elements contain name and class attributes. The Microcontainer uses these attributes to create an instance of the **callback** class, naming it so that it can be used as beans enter or leave the relevant state during deployment and undeployment. You can specify which beans are affected by the callback using the classes attribute, as shown in Example 5.5, “Using the classes Attribute”.

Example 5.5. Using the classes Attribute

```
<aop:lifecycle-install xmlns:aop="urn:jboss:aop-beans:1.0"

name="InstallAdvice"

class="org.jboss.test.microcontainer.support.LifecycleCallback"

classes="@org.jboss.test.microcontainer.support.Install">
```

```
</aop:lifecycle-install>
```

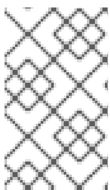
This code specifies that additional logic in the **lifecycleCallback** class is applied to any bean classes that are annotated with ***@org.jboss.test.microcontainer.support.Install*** before they enter and after they leave the **INSTALLED** state.

For the callback class to work, it must contain **install** and **uninstall** methods that take ***ControllerContext*** as a parameter, as shown in Example 5.6, “Install and Uninstall Methods”.

Example 5.6. Install and Uninstall Methods

```
import org.jboss.dependency.spi.ControllerContext;

public class LifecycleCallback {

    public void install(ControllerContext ctx) {
        System.out.println("Bean " + ctx.getName() + " is being
installed";
    }
    public void uninstall(ControllerContext ctx) {
     System.out.println("Bean " + ctx.getName() + " is being
uninstalled";
    }
}
```

The **install** method is called during the bean's deployment, and the **uninstall** method during its undeployment.



NOTE

Although behavior is being added to the deployment and undeployment process using callbacks, AOP is not actually used here. The *pointcut expression* functionality of JBoss AOP is used to determine which bean classes the behaviors apply to.

5.5. ADDING SERVICE LOOK-UPS THROUGH JNDI

Until now, you have used the Microcontainer to look up references to bean instances which represent services. This is not ideal, because it requires a reference to the Microcontainer kernel before the controller can be accessed. This is shown in Example 5.7, “Looking Up References To Beans”.

Example 5.7. Looking Up References To Beans

```
private HRManager manager;
private EmbeddedBootstrap bootstrap;
private Kernel kernel;
private KernelController controller;
private final static String HRSERVICE = "HRService";

...

// Start JBoss Microcontainer
bootstrap = new EmbeddedBootstrap();
bootstrap.run();

kernel = bootstrap.getKernel();
controller = kernel.getController();

...

 ControllerContext context = controller.getInstalledContext(HRSERVICE);
if (context != null) { manager = (HRManager) context.getTarget(); }
```

Handing out kernel references to every client that looks up a service is a security risk, because it provides wide-spread access to the Microcontainer configuration. For better security, apply the ServiceLocator pattern and use a class to performs look-ups on behalf of the clients. Even better, pass the bean references, along with their names, to the ServiceLocator at deployment time, using a lifecycle callback. In that scenario, the ServiceLocator can look them up without knowing about the Microcontainer at all. Undeployment would subsequently remove the bean references from the ServiceLocator to prevent further look-ups.

It would not be difficult to write your own ServiceLocator implementation. Integrating an existing one such as JBoss Naming Service (JBoss NS) is even quicker, and has the additional benefit of complying to the Java Naming and Directory Interface (JNDI) specification. JNDI enables clients to access different, possibly multiple, naming services using a common API.

Procedure 5.3. Writing Your Own ServiceLocator Implementation

1. First, create an instance of JBoss NS using the Microcontainer.
2. Next, add a lifecycle callback to perform the binding and unbinding of the bean references during deployment and undeployment.
3. Mark the bean classes you wish to bind references for, using annotations.
4. Now, you can locate the beans at run-time using the shorthand pointcut expression as shown earlier.

PART II. ADVANCED CONCEPTS WITH THE MICROCONTAINER

This section covers advanced concepts, and shows some interesting features of the Microcontainer. Code examples in the rest of the guide are assumed to be incomplete examples, and it is the programmer's responsibility to extrapolate and extend them as necessary.

CHAPTER 6. COMPONENT MODELS

The JBoss Microcontainer works within several popular POJO component models. Components are reusable software programs that you can develop and assemble easily to create sophisticated applications. Effective integration with these component models was a key goal for the Microcontainer. Some popular component models which can be used with the Microcontainer are JMX, Spring, and Guice.

6.1. ALLOWABLE INTERACTIONS WITH COMPONENT MODELS

Before discussing interaction with some of the popular component models, it is important to understand which types of interactions are allowable. JMX MBeans are one example of a component model. Their interactions include executing MBean operations, referencing attributes, setting attributes and declaring explicit dependencies between named MBeans.

The default behaviors and interactions in the Microcontainer are what you also normally get from any other *Inversion of Control (IoC)* container and are similar to the functionality provided by MBeans, including plain method invocations for operations, setters/getters for attributes and explicit dependencies.

6.2. A BEAN WITH NO DEPENDENCIES

Example 6.1, “Deployment Descriptor for a Plain POJO” shows a deployment descriptor for a plain POJO with no dependencies. This is the starting point for integrating the Microcontainer with Spring or Guice.

Example 6.1. Deployment Descriptor for a Plain POJO

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="PlainPojo" class="org.jboss.demos.models.plain.Pojo"/>

  <beanfactory name="PojoFactory"
class="org.jboss.demos.models.plain.Pojo">
    <property
name="factoryClass">org.jboss.demos.models.plain.PojoFactory</property>
  </beanfactory>

</deployment>
```

6.3. USING THE MICROCONTAINER WITH SPRING

Example 6.2. Descriptor with Spring Support

```
<beans xmlns="urn:jboss:spring-beans:2.0">

  <!-- Adding @Spring annotation handler -->
  <bean id="SpringAnnotationPlugin"
class="org.jboss.spring.annotations.SpringBeanAnnotationPlugin" />

  <bean id="SpringPojo" class="org.jboss.demos.models.spring.Pojo"/>

</beans>
```

This file's namespace is different from the plain Microcontainer bean's file. The **urn:jboss:spring-beans:2.0** namespace points to your version of the Spring schema port, which describes your bean's Spring style. The Microcontainer, rather than Spring's bean factory notion, deploys the beans.

Example 6.3. Using Spring with the Microcontainer

```
public class Pojo extends AbstractPojo implements BeanNameAware {
    private String beanName;

    public void setBeanName(String name)
    {
 beanName = name;
    }

    public String getBeanName()
    {
 return beanName;
    }

    public void start()
    {
 if ("SpringPojo".equals(getBeanName()) == false)
     throw new IllegalArgumentException("Name doesn't match: " +
getBeanName());
    }
}
```

Although the SpringPojo bean has a dependency on Spring's library caused by implementing BeanNameAware interface, its only purpose is to expose and mock some of the Spring's callback behavior.

6.4. USING GUICE WITH THE MICROCONTAINER

The focus of Guice is type matching. Guice beans are generated and configured using Modules.

Example 6.4. Deployment Descriptor for Guice Integration In the Microcontainer

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="GuicePlugin"
```

```
class="org.jboss.guice.spi.GuiceKernelRegistryEntryPlugin">
    <constructor>
      <parameter>
 <array elementClass="com.google.inject.Module">
   <bean class="org.jboss.demos.models.guice.PojoModule"/>
 </array>
      </parameter>
    </constructor>
  </bean>

</deployment>
```

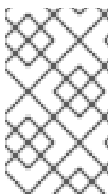
Two important parts to watch from this file are **PojoModule** and **GuiceKernelRegistryEntryPlugin**. **PojoModule** configures your beans, as in Example 6.5, “Configuring Beans for Guice”. **GuiceKernelRegistryEntryPlugin** provides integration with the Microcontainer, as shown in Example 6.6, “Guice Integration with the Microcontainer”.

Example 6.5. Configuring Beans for Guice

```
public class PojoModule extends AbstractModule {
    private Controller controller;

    @Constructor
 public PojoModule(@Inject(
      bean = KernelConstants.KERNEL_CONTROLLER_NAME)
     Controller controller)
    {
 this.controller = controller;
    }

    protected void configure()
    {
 bind(Controller.class).toInstance(controller);
 bind(IPojo.class).to(Pojo.class).in(Scopes.SINGLETON);
 bind(IPojo.class).annotatedWith(FromMC.class).
     toProvider(GuiceIntegration.fromMicrocontainer(IPojo.class,
"PlainPojo"));
    }
}
```

Example 6.6. Guice Integration with the Microcontainer

```
public class GuiceKernelRegistryEntryPlugin implements
KernelRegistryPlugin {
    private Injector injector;

    public GuiceKernelRegistryEntryPlugin(Module... modules)
    {
 injector = Guice.createInjector(modules);
    }

    public void destroy()
    {
 injector = null;
    }

    public KernelRegistryEntry getEntry(Object name)
    {
 KernelRegistryEntry entry = null;
 try
     {
  if (name instanceof Class<?>)
      {
   Class<?> clazz = (Class<?>)name;
   entry = new AbstractKernelRegistryEntry(name,
injector.getInstance(clazz));
      }
  else if (name instanceof Key)
      {
   Key<?> key = (Key<?>)name;
   entry = new AbstractKernelRegistryEntry(name,
injector.getInstance(key));
      }
     }
 catch (Exception ignored)
     {
     }
 return entry;
    }
}
```



NOTE

An **`Injector`** is created from the **`Modules`** class, then does a look-up on it for matching beans. See Section 6.5, “Legacy MBeans, and Mixing Different Component Models” for information about declaring and using legacy MBeans.

6.5. LEGACY MBEANS, AND MIXING DIFFERENT COMPONENT MODELS

The simplest example of mixing different content models is shown in Example 6.7, “Injecting a POJO Into an MBean”.

Example 6.7. Injecting a POJO Into an MBean

```
<server>

  <mbean code="org.jboss.demos.models.mbeans.Pojo"
name="jboss.demos:service=pojo">
    <attribute name="OtherPojo"><inject bean="PlainPojo"/></attribute>
  </mbean>

</server>
```

To deploy MBeans deployment via the Microcontainer, you must write an entirely new handler for the component model. See **`system-jmx-beans.xml`** for more details. The code from this file lives in the JBoss Application Server source code: system-jmx sub-project.

6.6. EXPOSING POJOS AS MBEANS

Example 6.8. Exposing an Existing POJO as an MBean

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="AnnotatedJMXPojo"
class="org.jboss.demos.models.jmx.AtJmxPojo"/>

  <bean name="XmlJMXPojo" class="org.jboss.demos.models.mbeans.Pojo">

<annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX(exposedInterfa
ce=org.jboss.demos.models.mbeans.PojoMBean.class,
registerDirectly=true)</annotation>
  </bean>

  <bean name="ExposedPojo" class="org.jboss.demos.models.jmx.Pojo"/>

  <bean name="AnnotatedExposePojo"
class="org.jboss.demos.models.jmx.ExposePojo">
    <constructor>
      <parameter><inject bean="ExposedPojo"/></parameter>
    </constructor>
  </bean>

</deployment>
```

This descriptor exposes an existing POJO as an MBean, and registers it into an MBean server.

To expose a POJO as an MBean, end it with an @JMX annotation, assuming that you have imported **org.jboss.aop.microcontainer.aspects.jmx.JMX**. The bean can either be exposed directly, or in its property.

Example 6.9. Exposing a POJO as an MBean Using Its Property

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="XMLLoginConfig"
class="org.jboss.demos.models.old.XMLLoginConfig"/>

  <bean name="SecurityConfig"
class="org.jboss.demos.models.old.SecurityConfig">
    <property name="defaultLoginConfig"><inject
bean="XMLLoginConfig"/></property>
  </bean>

  <bean name="SecurityChecker"
class="org.jboss.demos.models.old.Checker">
    <property name="loginConfig"><inject
bean="jboss.security:service=XMLLoginConfig"/></property>
    <property name="securityConfig"><inject
bean="jboss.security:service=SecurityConfig"/></property>
  </bean>

</deployment>
```

You can use any of the injection look-up types, by either looking up a plain POJO or getting a handle to an MBean from the MBean server. One of the injection options is to use type injection, sometimes called *autowiring*, and shown in Example 6.10, “Autowiring”.

Example 6.10. Autowiring

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <bean name="FromGuice"
class="org.jboss.demos.models.plain.FromGuice">
    <constructor><parameter><inject
bean="PlainPojo"/></parameter></constructor>
    <property name="guicePojo"><inject/></property>
  </bean>

  <bean name="AllPojos" class="org.jboss.demos.models.plain.AllPojos">
    <property name="directMBean"><inject
bean="jboss.demos:service=pojo"/></property>
    <property name="exposedMBean"><inject
bean="jboss.demos:service=ExposedPojo"/></property>
    <property name="exposedMBean"><inject
bean="jboss.demos:service=ExposedPojo"/></property>
```

```
  </bean>

</deployment>
```

The FromGuice bean injects the Guice bean via type matching, where **`PlainPojo`** is injected with a common name injection. Now, you can test if Guice binding works as expected, as shown in Example 6.11, “Testing Guice Functionality”.

Example 6.11. Testing Guice Functionality

```
public class FromGuice {
    private IPojo plainPojo;
    private org.jboss.demos.models.guice.Pojo guicePojo;

    public FromGuice(IPojo plainPojo)
    {
 this.plainPojo = plainPojo;
    }

    public void setGuicePojo(org.jboss.demos.models.guice.Pojo
guicePojo)
    {
 this.guicePojo = guicePojo;
    }

    public void start()
    {
 f (plainPojo != guicePojo.getMcPojo())
     throw new IllegalArgumentException("Pojos are not the same: " +
plainPojo + "!=" + guicePojo.getMcPojo());
    }
}
```

Example 6.11, “Testing Guice Functionality” only provides an alias component model. The alias is a trivial, but necessary feature. It must be introduced as a new component model inside the Microcontainer, in order to implement it as a true dependency. The implementation details are shown in Example 6.12, “AbstractController Source Code”.

Example 6.12. AbstractController Source Code

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">
  <alias name="SpringPojo">springPojo</alias>
```

```
</deployment>
```

This descriptor maps the SpringPojo name to the springPojo alias. The benefit of aliases as true component models is that timing of bean deployment becomes less important. The alias waits in a non-installed state until the real bean triggers it.

CHAPTER 7. ADVANCED DEPENDENCY INJECTION AND IOC

Today, *Dependency injection (DI)*, also called *Inversion of Control (IoC)*, lies at the core of many frameworks that embrace the notion of a container or a component model. Component models were discussed in a previous chapter. JBoss JMX kernel, the precursor to the Microcontainer, provided only lightweight DI/IoC support, primarily due to the limitations of accessing MBeans through the MBeans server. However with the new POJO-based component model, several new and interesting features are available.

This chapter shows how you can apply different DI concepts with the help of the JBoss Microcontainer. These concepts will be expressed via XML code, but you can also apply most of these features using annotations.

7.1. VALUE FACTORY

A value factory is a bean which has one or more methods devoted to generating values for you. See Example 7.1, “Value Factory”.

Example 7.1. Value Factory

```
<bean name="Binding" class="org.jboss.demos.ioc.vf.PortBindingManager">
  <constructor>
    <parameter>
      <map keyClass="java.lang.String" valueClass="java.lang.Integer">
 <entry>
   <key>http</key>
   <value>80</value>
 </entry>
 <entry>
   <key>ssh</key>
   <value>22</value>
 </entry>
      </map>
    </parameter>
  </constructor>
</bean>
<bean name="PortsConfig" class="org.jboss.demos.ioc.vf.PortsConfig">
  <property name="http"><value-factory bean="Binding" method="getPort"
parameter="http"/></property>
  <property name="ssh"><value-factory bean="Binding" method="getPort"
parameter="ssh"/></property>
  <property name="ftp">
    <value-factory bean="Binding" method="getPort">
      <parameter>ftp</parameter>
      <parameter>21</parameter>
    </value-factory>
  </property>
  <property name="mail">
    <value-factory bean="Binding" method="getPort">
      <parameter>mail</parameter>
      <parameter>25</parameter>
    </value-factory>
  </property>
</bean>
```

Example 7.2, “PortsConfig” shows how the PortsConfig bean uses Binding bean to get its values via the **getPort** method invocation.

Example 7.2. PortsConfig

```
public class PortBindingManager {
    private Map<String, Integer> bindings;
    public PortBindingManager(Map<String, Integer> bindings)
    {
 this.bindings = bindings;
    }
    public Integer getPort(String key)
    {
 return getPort(key, null);
    }
    public Integer getPort(String key, Integer defaultValue)
    {
 if (bindings == null)
     return defaultValue;
 Integer value = bindings.get(key);
 if (value != null)
     return value;
 if (defaultValue != null)
     bindings.put(key, defaultValue);
 return defaultValue;
    }
}
```

7.2. CALLBACKS

The descriptor shown in Example 7.3, “Callbacks to Collect and Filter Beans” allows you to collect all beans of a certain type, and even limit the number of matching beans.

In conjunction with the descriptor in Example 7.3, “Callbacks to Collect and Filter Beans”, the Java code shown in Example 7.4, “A Parser to Collect all Editors” shows a **Parser** which collects all **Editor**s.

Example 7.3. Callbacks to Collect and Filter Beans

```
<bean name="checker" class="org.jboss.demos.ioc.callback.Checker">
  <constructor>
    <parameter>
      <value-factory bean="parser" method="parse">
```

```
  <parameter>
    <array elementClass="java.lang.Object">
      <value>http://www.jboss.org</value>
      <value>SI</value>
      <value>3.14</value>
      <value>42</value>
    </array>
  </parameter>
        </value-factory>
      </parameter>
   </constructor>
</bean>
<bean name="editorA" class="org.jboss.demos.ioc.callback.DoubleEditor"/>
<bean name="editorB" class="org.jboss.demos.ioc.callback.LocaleEditor"/>
<bean name="parser" class="org.jboss.demos.ioc.callback.Parser">
  <incallback method="addEditor" cardinality="4..n"/>
  <uncallback method="removeEditor"/>
</bean>
<bean name="editorC" class="org.jboss.demos.ioc.callback.LongEditor"/>
<bean name="editorD" class="org.jboss.demos.ioc.callback.URLEditor"/>
```

Example 7.4. A Parser to Collect all Editors

```
public class Parser {
    private Set<Editor> editors = new HashSet<Editor>();
    ...
 public void addEditor(Editor editor)
    {
 editors.add(editor);
    }
    public void removeEditor(Editor editor)
    {
 editors.remove(editor);
    }
}
```

Notice that **incallback** and **uncallback** use the method name for matching.

```
<incallback method="addEditor" cardinality="4..n"/>
<uncallback method="removeEditor"/>
```

A bottom limit controls how many editors actually cause the bean to progress from a Configured state: **`cardinality=4..n/>`**

Eventually, **`Checker`** gets created and checks the parser. This is illustrated in Example 7.5, “The Checker for the Parser”.

Example 7.5. The Checker for the Parser

```
public void create() throws Throwable {
    Set<String> strings = new TreeSet<String>
(String.CASE_INSENSITIVE_ORDER);
    for (Object element : elements)
 strings.add(element.toString());
    if (expected.equals(strings) == false)
 throw new IllegalArgumentException("Illegal expected set: " + expected
+ "!=" + strings);
}
```

7.3. BEAN ACCESS MODE

With the default BeanAccessMode, a bean's fields are not inspected. However, if you specify a different BeanAccessMode, the fields are accessible as part of the bean's properties. See Example 7.6, “Possible BeanAccessMode Definitions”, Example 7.7, “Setting the BeanAccessMode”, and Example 7.8, “The **`FieldsBean`** Class” for an implementation.

Example 7.6. Possible BeanAccessMode Definitions

```
public enum BeanAccessMode {
    STANDARD(BeanInfoCreator.STANDARD), // Getters and Setters
 FIELDS(BeanInfoCreator.FIELDS), // Getters/Setters and fields without
getters and setters
 ALL(BeanInfoCreator.ALL); // As above but with non public fields
included
}
```

Here, a String value is set to a private String field:

Example 7.7. Setting the BeanAccessMode

```
<bean name="FieldsBean" class="org.jboss.demos.ioc.access.FieldsBean"
access-mode="ALL">
  <property name="string">InternalString</property>
</bean>
```

Example 7.8. The FieldsBean Class

```
public class FieldsBean {
    private String string;
    public void start()
    {
 if (string == null)
     throw new IllegalArgumentException("Strings should be set!");
    }
}
```

7.4. BEAN ALIAS

Each bean can have any number of aliases. Since Microcontainer component names are treated as Objects, the alias type is not limited. By default a system property replacement is not done; you need to set the replace flag explicitly, as shown in Example 7.9, “A Simple Bean Alias”.

Example 7.9. A Simple Bean Alias

```
<bean name="SimpleName" class="java.lang.Object">
  <alias>SimpleAlias</alias>
  <alias replace="true">${some.system.property}</alias>
  <alias class="java.lang.Integer">12345</alias>
  <alias><javabean xmlns="urn:jboss:javabean:2.0"
class="org.jboss.demos.bootstrap.Main"/></alias>
</bean>
```

7.5. XML (OR METADATA) ANNOTATIONS SUPPORT

AOP support is a prime feature in JBoss Microcontainer. You can use AOP aspects and plain beans in any combination. Example 7.10, “Intercepting a Method based on Annotation” attempts to intercept a method invocation based on an annotation. The annotation can come from anywhere. It might be a true class annotation or an annotation added through the xml configuration.

Example 7.10. Intercepting a Method based on Annotation

```
<interceptor xmlns="urn:jboss:aop-beans:1.0" name="StopWatchInterceptor"
class="org.jboss.demos.ioc.annotations.StopWatchInterceptor"/>

<bind xmlns="urn:jboss:aop-beans:1.0" pointcut="execution(*
@org.jboss.demos.ioc.annotations.StopWatchLog->*(..)) OR execution(* *-
>@org.jboss.demos.ioc.annotations.StopWatchLog(..))">
  <interceptor-ref name="StopWatchInterceptor"/>
</bind>
</interceptor>
```

```
public class StopWatchInterceptor implements Interceptor {
    ...
 public Object invoke(Invocation invocation) throws Throwable
    {
 Object target = invocation.getTargetObject();
 long time = System.currentTimeMillis();
 log.info("Invocation [" + target + "] start: " + time);
 try
     {
  return invocation.invokeNext();
     }
 finally
     {
  log.info("Invocation [" + target + "] time: " +
(System.currentTimeMillis() - time));
     }
    }
}
```

Example 7.11, “A true class annotated executor” and Example 7.12, “Simple executor with XML annotation” show some different ways to implement executors.

Example 7.11. A true class annotated executor

```
<bean name="AnnotatedExecutor"
class="org.jboss.demos.ioc.annotations.AnnotatedExecutor">
```

```
public class AnnotatedExecutor implements Executor {
    ...
 @StopWatchLog // <-- Pointcut match!
     public void execute() throws Exception {
 delegate.execute();
    }
}
```

Example 7.12. Simple executor with XML annotation

```
<bean name="SimpleExecutor"
class="org.jboss.demos.ioc.annotations.SimpleExecutor">
  <annotation>@org.jboss.demos.ioc.annotations.StopWatchLog</annotation>
// <-- Pointcut match!
</bean>
```

```
public class SimpleExecutor implements Executor {
    private static Random random = new Random();
    public void execute() throws Exception
    {
 Thread.sleep(Math.abs(random.nextLong() % 101));
    }
}
```

After adding executor invoker beans, you can see the executors in action during employment, by looking for log output such as Example 7.13, “Executor Logging Output”.

Example 7.13. Executor Logging Output

```
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.AnnotatedExecutor@4d28c7] start:
1229345859234
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.AnnotatedExecutor@4d28c7] time: 31
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.SimpleExecutor@1b044df] start:
1229345859265
    JBoss-MC-Demo  INFO [15-12-2008 13:57:39] StopWatch - Invocation
[org.jboss.demos.ioc.annotations.SimpleExecutor@1b044df] time: 47
```

7.6. AUTOWIRE

Autowiring, or contextual injection, is a common feature with IoC frameworks. Example 7.14, “Include and Exclude with Autowiring” shows you how to use or exclude beans with autowiring.

Example 7.14. Include and Exclude with Autowiring

```
<bean name="Square" class="org.jboss.demos.ioc.autowire.Square"
autowire-candidate="false"/>
<bean name="Circle" class="org.jboss.demos.ioc.autowire.Circle"/>
<bean name="ShapeUser" class="org.jboss.demos.ioc.autowire.ShapeUser">
  <constructor>
    <parameter><inject/></parameter>
  </constructor>
</bean>
<bean name="ShapeHolder"
class="org.jboss.demos.ioc.autowire.ShapeHolder">
  <incallback method="addShape"/>
  <uncallback method="removeShape"/>
</bean>
<bean name="ShapeChecker"
class="org.jboss.demos.ioc.autowire.ShapesChecker"/>
```

In both cases - ShapeUser and ShapeChecker - only Circle should be used, since Square is excluded in the contextual binding.

7.7. BEAN FACTORY

When you want more than one instance of a particular bean, you need to use the bean factory pattern. The job of the Microcontainer is to configure and install the bean factory as if it were a plain bean. Then you need to invoke the bean factory's **`createBean`** method.

By default, the Microcontainer creates a **`GenericBeanFactory`** instance, but you can configure your own factory. The only limitation is that its signature and configuration hooks are similar to the one of **`AbstractBeanFactory`**.

Example 7.15. Generic Bean Factory

```
<bean name="Object" class="java.lang.Object"/>
<beanfactory name="DefaultPrototype"
```

```
class="org.jboss.demos.ioc.factory.Prototype">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<beanfactory name="EnhancedPrototype"
class="org.jboss.demos.ioc.factory.Prototype"
factoryClass="org.jboss.demos.ioc.factory.EnhancedBeanFactory">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<beanfactory name="ProxiedPrototype"
class="org.jboss.demos.ioc.factory.UnmodifiablePrototype"
factoryClass="org.jboss.demos.ioc.factory.EnhancedBeanFactory">
  <property name="value"><inject bean="Object"/></property>
</beanfactory>
<bean name="PrototypeCreator"
class="org.jboss.demos.ioc.factory.PrototypeCreator">
  <property name="default"><inject bean="DefaultPrototype"/></property>
  <property name="enhanced"><inject
bean="EnhancedPrototype"/></property>
  <property name="proxied"><inject bean="ProxiedPrototype"/></property>
</bean>
```

See Example 7.16, “Extended BeanFactory” for usage of an extended BeanFactory.

Example 7.16. Extended BeanFactory

```
public class EnhancedBeanFactory extends GenericBeanFactory {
    public EnhancedBeanFactory(KernelConfigurator configurator)
    {
 super(configurator);
    }
    public Object createBean() throws Throwable
    {
 Object bean = super.createBean();
 Class clazz = bean.getClass();
 if (clazz.isAnnotationPresent(SetterProxy.class))
     {
  Set<Class> interfaces = new HashSet<Class>();
  addInterfaces(clazz, interfaces);
  return Proxy.newProxyInstance(
          clazz.getClassLoader(),
          interfaces.toArray(new Class[interfaces.size()]),
          new SetterInterceptor(bean)
          );
     }
 else
     {
  return bean;
     }
    }
```

```
    protected static void addInterfaces(Class clazz, Set<Class>
interfaces)
    {
 if (clazz == null)
     return;
 interfaces.addAll(Arrays.asList(clazz.getInterfaces()));
 addInterfaces(clazz.getSuperclass(), interfaces);
    }
    private class SetterInterceptor implements InvocationHandler
    {
 private Object target;
 private SetterInterceptor(Object target)
 {
     this.target = target;
 }
 public Object invoke(Object proxy, Method method, Object[] args) throws
Throwable
 {
     String methodName = method.getName();
     if (methodName.startsWith("set"))
  throw new IllegalArgumentException("Cannot invoke setters.");
     return method.invoke(target, args);
 }
    }
}
```

```
public class PrototypeCreator {
    ...
 public void create() throws Throwable
    {
 ValueInvoker vi1 = (ValueInvoker)bfDefault.createBean();
 vi1.setValue("default");
 ValueInvoker vi2 = (ValueInvoker)enhanced.createBean();
 vi2.setValue("enhanced");
 ValueInvoker vi3 = (ValueInvoker)proxied.createBean();
 try
     {
  vi3.setValue("default");
  throw new Error("Should not be here.");
     }
 catch (Exception ignored)
     {
     }
    }
```

7.8. BEAN METADATA BUILDER

When using the Microcontainer in your code, use **`BeanMetaDataBuilder`** to create and configure your bean metadata.

Example 7.17. `BeanMetaDataBuilder`

```
<bean name="BuilderUtil"
```

```
class="org.jboss.demos.ioc.builder.BuilderUtil"/>
<bean name="BuilderExampleHolder"
class="org.jboss.demos.ioc.builder.BuilderExampleHolder">
  <constructor>
    <parameter><inject bean="BUExample"/></parameter>
  </constructor>
</bean>
```

Using this concept, you don't expose your code to any Microcontainer implementation details.

```
public class BuilderUtil {
    private KernelController controller;
    @Constructor
 public BuilderUtil(@Inject(bean =
KernelConstants.KERNEL_CONTROLLER_NAME) KernelController controller) {
 this.controller = controller;
    }
    public void create() throws Throwable {
 BeanMetaDataBuilder builder =
BeanMetaDataBuilder.createBuilder("BUExample",
BuilderExample.class.getName());
 builder.addStartParameter(Kernel.class.getName(),
builder.createInject(KernelConstants.KERNEL_NAME));
 controller.install(builder.getBeanMetaData());
    }
    public void destroy() {
 controller.uninstall("BUExample");
    }
}
```

7.9. CUSTOM CLASSLOADER

In the Microcontainer you can define a custom ClassLoader per bean. When defining a classloader for the whole deployment, make sure you don't create a cyclic dependency -- for instance, a newly defined classloader that depends on itself.

Example 7.18. Defining a ClassLoader Per Bean

```
<classloader><inject bean="custom-classloader:0.0.0"/></classloader>
<!-- this will be explained in future article -->
<classloader name="custom-classloader" xmlns="urn:jboss:classloader:1.0"
export-all="NON_EMPTY" import-all="true"/>
<bean name="CustomCL"
class="org.jboss.demos.ioc.classloader.CustomClassLoader">
  <constructor>
    <parameter><inject bean="custom-classloader:0.0.0"/></parameter>
  </constructor>
  <property name="pattern">org\.jboss\.demos\.ioc\..+</property>
```

```
</bean>
<bean name="CB1" class="org.jboss.demos.ioc.classloader.CustomBean"/>
<bean name="CB2" class="org.jboss.demos.ioc.classloader.CustomBean">
  <classloader><inject bean="CustomCL"/></classloader>
</bean>
```

Example 7.19, “Custom ClassLoader Test” shows a test to verify that the CB2 bean uses a custom ClassLoader, which limits the loadable package scope.

Example 7.19. Custom ClassLoader Test

```
public class CustomClassLoader extends ClassLoader {
    private Pattern pattern;
    public CustomClassLoader(ClassLoader parent) {
 super(parent);
    }
    public Class<?> loadClass(String name) throws
ClassNotFoundException {
 if (pattern == null || pattern.matcher(name).matches())
     return super.loadClass(name);
 else
     throw new ClassNotFoundException("Name '" + name + "' doesn't
match pattern: " + pattern);
    }
    public void setPattern(String regexp) {
 pattern = Pattern.compile(regexp);
    }
}
```

7.10. CONTROLLER MODE

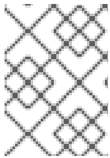
By default, the Microcontainer uses the AUTO controller mode. It pushes beans as far as they go with respect to dependencies. But there are two other modes: MANUAL and ON_DEMAND.

If the bean is marked as ON_DEMAND, it will not be used or installed installed until some other bean explicitly depends on it. In MANUAL mode, the Microcontainer user must push the bean forward and backward along the state ladder.

Example 7.20. Bean Controller Mode

```
<bean name="OptionalService"
class="org.jboss.demos.ioc.mode.OptionalService" mode="On Demand"/>
```

```
<bean name="OptionalServiceUser"
class="org.jboss.demos.ioc.mode.OptionalServiceUser"/>
<bean name="ManualService"
class="org.jboss.demos.ioc.mode.ManualService" mode="Manual"/>
<bean name="ManualServiceUser"
class="org.jboss.demos.ioc.mode.ManualServiceUser">
  <start>
    <parameter><inject bean="ManualService" fromContext="context"
state="Not Installed"/></parameter>
  </start>
</bean>
```



NOTE

Using the fromContext attribute of the **`inject`** class, you can inject beans, as well as their unmodifiable Microcontainer component representation.

Review the code of **`OptionalServiceUser`** and **`ManualServiceUser`** for how to use the Microcontainer API for ON_DEMAND and MANUAL bean handling.

7.11. CYCLE

Beans may depend on each other in a cycle. For instance, A depends on B at construction, but B depends on A at setter. Because of the Microcontainer's fine-grained state lifecycle separation, this problem can be solved easily.

Example 7.21. Bean Lifecycle Separation

```
<bean name="cycleA" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleB"/></property>
</bean>
<bean name="cycleB" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <constructor><parameter><inject bean="cycleA"
state="Instantiated"/></parameter></constructor>
</bean>
<bean name="cycleC" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleD"/></property>
</bean>
<bean name="cycleD" class="org.jboss.demos.ioc.cycle.CyclePojo">
  <property name="dependency"><inject bean="cycleC"
state="Instantiated"/></property>
</bean>
```

7.12. DEMAND AND SUPPLY

Sometimes, such as with an injection, a dependency between two beans may not be readily apparent. Such dependencies should be expressed in a clear way, such as shown in Example 7.22, “Static Code Usage”.

Example 7.22. Static Code Usage

```
<bean name="TMDemand"
class="org.jboss.demos.ioc.demandsupply.TMDemander">
  <demand>TM</demand>
</bean>
<bean name="SimpleTMSupply"
class="org.jboss.demos.ioc.demandsupply.SimpleTMSupplyer">
  <supply>TM</supply>
</bean>
```

7.13. INSTALLS

As a bean moves through different states, you might want to invoke some methods on other beans or the same bean. Example 7.23, “Invoking Methods in Different-States” shows how **`Entry`** invokes **`RepositoryManager`**'s **`add`** and **`removeEntry`** methods to register and unregister itself.

Example 7.23. Invoking Methods in Different-States

```
<bean name="RepositoryManager"
class="org.jboss.demos.ioc.install.RepositoryManager">
  <install method="addEntry">
    <parameter><inject fromContext="name"/></parameter>
    <parameter><this/></parameter>
  </install>
  <uninstall method="removeEntry">
    <parameter><inject fromContext="name"/></parameter>
  </uninstall>
</bean>
<bean name="Entry" class="org.jboss.demos.ioc.install.SimpleEntry">
  <install bean="RepositoryManager" method="addEntry"
state="Instantiated">
    <parameter><inject fromContext="name"/></parameter>
    <parameter><this/></parameter>
  </install>
  <uninstall bean="RepositoryManager" method="removeEntry"
state="Configured">
    <parameter><inject fromContext="name"/></parameter>
  </uninstall>
</bean>
```

7.14. LAZY MOCK

You might have a dependency on a bean that is rarely used, but takes a long time to configure. You can use a lazy mock of the bean, demonstrated in Example 7.24, “Lazy Mock”, to resolve the dependency. When you actually need the bean, invoke and use the target bean, hoping it has been installed by then.

Example 7.24. Lazy Mock

```
<bean name="lazyA" class="org.jboss.demos.ioc.lazy.LazyImpl">
  <constructor>
    <parameter>
      <lazy bean="lazyB">
 <interface>org.jboss.demos.ioc.lazy.ILazyPojo</interface>
      </lazy>
    </parameter>
  </constructor>
</bean>
<bean name="lazyB" class="org.jboss.demos.ioc.lazy.LazyImpl">
  <constructor>
    <parameter>
      <lazy bean="lazyA">
 <interface>org.jboss.demos.ioc.lazy.ILazyPojo</interface>
      </lazy>
    </parameter>
  </constructor>
</bean>
<lazy name="anotherLazy" bean="Pojo" exposeClass="true"/>
<bean name="Pojo" class="org.jboss.demos.ioc.lazy.Pojo"/>
```

7.15. LIFECYCLE

By default the Microcontainer uses **`create`**, **`start`**, and **`destroy`** methods when it moves through the various states. However, you may not want the Microcontainer to invoke them. For this reason, an ignore flag is available.

Example 7.25. Bean Lifecycles

```
<bean name="FullLifecycleBean-3"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean"/>
<bean name="FullLifecycleBean-2"
```

```
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean">
  <create ignored="true"/>
</bean>
<bean name="FullLifecycleBean-1"
class="org.jboss.demos.ioc.lifecycle.FullLifecycleBean">
  <start ignored="true"/>
</bean>
```

CHAPTER 8. THE VIRTUAL FILE SYSTEM

Duplication of resource-handling code is a common problem for developers. In most cases, the code deals with determining information about a particular resource, which might be a file, a directory, or, in the case of a JAR, a remote URL. Another duplication problem is code for the processing of nested archives. Example 8.1, “Resource Duplication Problem” illustrates the problem.

Example 8.1. Resource Duplication Problem

```
public static URL[] search(ClassLoader cl, String prefix, String suffix)
throws IOException {
    Enumeration[] e = new Enumeration[]{
 cl.getResources(prefix),
 cl.getResources(prefix + "MANIFEST.MF")
    };
    Set all = new LinkedHashSet();
    URL url;
    URLConnection conn;
    JarFile jarFile;
    for (int i = 0, s = e.length; i < s; ++i)
 {
     while (e[i].hasMoreElements())
  {
      url = (URL)e[i].nextElement();
      conn = url.openConnection();
      conn.setUseCaches(false);
      conn.setDefaultUseCaches(false);
      if (conn instanceof JarURLConnection)
   {
       jarFile = ((JarURLConnection)conn).getJarFile();
   }
      else
   {
       jarFile = getAlternativeJarFile(url);
   }
      if (jarFile != null)
   {
       searchJar(cl, all, jarFile, prefix, suffix);
   }
      else
   {
       boolean searchDone = searchDir(all, new
File(URLDecoder.decode(url.getFile(), "UTF-8")), suffix);
       if (searchDone == false)
    {
        searchFromURL(all, prefix, suffix, url);
    }
   }
  }
 }
    return (URL[])all.toArray(new URL[all.size()]);
}

private static boolean searchDir(Set result, File file, String suffix)
throws IOException
{
```

```
      if (file.exists() && file.isDirectory())
   {
       File[] fc = file.listFiles();
       String path;
       for (int i = 0; i < fc.length; i++)
    {
        path = fc[i].getAbsolutePath();
        if (fc[i].isDirectory())
     {
         searchDir(result, fc[i], suffix);
     }
        else if (path.endsWith(suffix))
     {
         result.add(fc[i].toURL());
     }
    }
       return true;
   }
      return false;
  }
```

There are also many problems with file locking on Windows systems, forcing developers to copy all hot-deployable archives to another location to prevent locking those in deploy folders (which would prevent their deletion and file-system based undeploy). File locking is a major problem whose only solution used to be centralizing all the resource loading code in one place.

The *VFS* project was created solve all of these issues. VFS stands for *Virtual File System*.

8.1. VFS PUBLIC API

VFS is used for two main purposes, as shown in Uses for VFS.

Uses for VFS

- simple resource navigation
- visitor pattern *API* (Application Programmer Interface)

As mentioned, in plain JDK, handling and navigating resources are complex. You must always check the resource type, and these checks can be cumbersome. VFS abstracts resources into a single resource type, VirtualFile.

Example 8.2. The VirtualFile Resource Type

```
public class VirtualFile implements Serializable {
    /**
     * Get certificates.
     *
     * @return the certificates associated with this virtual file
     */
    Certificate[] getCertificates()

    /**
     * Get the simple VF name (X.java)
```

```
      *
      * @return the simple file name
      * @throws IllegalStateException if the file is closed
      */
 String getName()

    /**
      * Get the VFS relative path name (org/jboss/X.java)
      *
      * @return the VFS relative path name
      * @throws IllegalStateException if the file is closed
      */
 String getPathName()

    /**
      * Get the VF URL (file://root/org/jboss/X.java)
      *
      * @return the full URL to the VF in the VFS.
      * @throws MalformedURLException if a url cannot be parsed
      * @throws URISyntaxException if a uri cannot be parsed
      * @throws IllegalStateException if the file is closed
      */
 URL toURL() throws MalformedURLException, URISyntaxException

    /**
      * Get the VF URI (file://root/org/jboss/X.java)
      *
      * @return the full URI to the VF in the VFS.
      * @throws URISyntaxException if a uri cannot be parsed
      * @throws IllegalStateException if the file is closed
      * @throws MalformedURLException for a bad url
      */
       URI toURI() throws MalformedURLException, URISyntaxException

    /**
      * When the file was last modified
      *
      * @return the last modified time
      * @throws IOException for any problem accessing the virtual file
system
      * @throws IllegalStateException if the file is closed
      */
       long getLastModified() throws IOException

    /**
      * Returns true if the file has been modified since this method was
last called
      * Last modified time is initialized at handler instantiation.
      *
      * @return true if modifed, false otherwise
      * @throws IOException for any error
      */
       boolean hasBeenModified() throws IOException

    /**
      * Get the size
```

```
    *
    * @return the size
    * @throws IOException for any problem accessing the virtual file
system
    * @throws IllegalStateException if the file is closed
    */
     long getSize() throws IOException

   /**
    * Tests whether the underlying implementation file still exists.
    * @return true if the file exists, false otherwise.
    * @throws IOException - thrown on failure to detect existence.
    */
     boolean exists() throws IOException

   /**
    * Whether it is a simple leaf of the VFS,
    * i.e. whether it can contain other files
    *
    * @return true if a simple file.
    * @throws IOException for any problem accessing the virtual file
system
    * @throws IllegalStateException if the file is closed
    */
     boolean isLeaf() throws IOException

   /**
    * Is the file archive.
    *
    * @return true if archive, false otherwise
    * @throws IOException for any error
    */
     boolean isArchive() throws IOException

   /**
    * Whether it is hidden
    *
    * @return true when hidden
    * @throws IOException for any problem accessing the virtual file
system
    * @throws IllegalStateException if the file is closed
    */
     boolean isHidden() throws IOException

   /**
    * Access the file contents.
    *
    * @return an InputStream for the file contents.
    * @throws IOException for any error accessing the file system
    * @throws IllegalStateException if the file is closed
    */
     InputStream openStream() throws IOException

   /**
    * Do file cleanup.
    *
```

```
       * e.g. delete temp files
       */
       void cleanup()

      /**
       * Close the file resources (stream, etc.)
       */
       void close()

      /**
       * Delete this virtual file
       *
       * @return true if file was deleted
       * @throws IOException if an error occurs
       */
       boolean delete() throws IOException

      /**
       * Delete this virtual file
       *
       * @param gracePeriod max time to wait for any locks (in
milliseconds)
       * @return true if file was deleted
       * @throws IOException if an error occurs
       */
       boolean delete(int gracePeriod) throws IOException

      /**
       * Get the VFS instance for this virtual file
       *
       * @return the VFS
       * @throws IllegalStateException if the file is closed
       */
       VFS getVFS()

      /**
       * Get the parent
       *
       * @return the parent or null if there is no parent
       * @throws IOException for any problem accessing the virtual file
system
       * @throws IllegalStateException if the file is closed
       */
       VirtualFile getParent() throws IOException

      /**
       * Get a child
       *
       * @param path the path
       * @return the child or <code>null</code> if not found
       * @throws IOException for any problem accessing the VFS
       * @throws IllegalArgumentException if the path is null
       * @throws IllegalStateException if the file is closed or it is a
leaf node
       */
       VirtualFile getChild(String path) throws IOException
```

```
    /**
     * Get the children
     *
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
      List<VirtualFile> getChildren() throws IOException

    /**
     * Get the children
     *
     * @param filter to filter the children
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed or it is a
leaf node
     */
      List<VirtualFile> getChildren(VirtualFileFilter filter) throws
IOException

    /**
     * Get all the children recursively<p>
     *
     * This always uses {@link VisitorAttributes#RECURSE}
     *
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed
     */
      List<VirtualFile> getChildrenRecursively() throws IOException

    /**
     * Get all the children recursively<p>
     *
     * This always uses {@link VisitorAttributes#RECURSE}
     *
     * @param filter to filter the children
     * @return the children
     * @throws IOException for any problem accessing the virtual file
system
     * @throws IllegalStateException if the file is closed or it is a
leaf node
     */
      List<VirtualFile> getChildrenRecursively(VirtualFileFilter
filter) throws IOException

    /**
     * Visit the virtual file system
     *
     * @param visitor the visitor
     * @throws IOException for any problem accessing the virtual file
```

```
system
      * @throws IllegalArgumentException if the visitor is null
      * @throws IllegalStateException if the file is closed
      */
       void visit(VirtualFileVisitor visitor) throws IOException
       }
```

All of the usual read-only File System operations are available, plus a few options to cleanup or delete the resource. Cleanup or deletion handling is needed when dealing with some internal temporary files, such as files created to handle nested jars.

To switch from the JDK's File or URL resource handling to new VirtualFile you need a root VirtualFile, which is provided by the **VFS** class, with the help of URL or URI parameter.

Example 8.3. Using the VFS Class

```
public class VFS {
    /**
     * Get the virtual file system for a root uri
     *
     * @param rootURI the root URI
     * @return the virtual file system
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VFS getVFS(URI rootURI) throws IOException

    /**
     * Create new root
     *
     * @param rootURI the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile createNewRoot(URI rootURI) throws IOException

    /**
     * Get the root virtual file
     *
     * @param rootURI the root uri
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
     */
    static VirtualFile getRoot(URI rootURI) throws IOException

    /**
     * Get the virtual file system for a root url
     *
     * @param rootURL the root url
     * @return the virtual file system
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL is null
```

```
     */
    static VFS getVFS(URL rootURL) throws IOException

    /**
     * Create new root
     *
     * @param rootURL the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile createNewRoot(URL rootURL) throws IOException

    /**
     * Get the root virtual file
     *
     * @param rootURL the root url
     * @return the virtual file
     * @throws IOException if there is a problem accessing the VFS
     * @throws IllegalArgumentException if the rootURL
     */
    static VirtualFile getRoot(URL rootURL) throws IOException

    /**
     * Get the root file of this VFS
     *
     * @return the root
     * @throws IOException for any problem accessing the VFS
     */
    VirtualFile getRoot() throws IOException
}
```

The three different methods look similar.

- **`getVFS`**
- **`createNewRoot`**
- **`getRoot`**

`getVFS` returns a VFS instance but does not yet create a VirtualFile instance. This is important because there are methods which help with configuring a VFS instance (see VFS class API javadocs), before instructing it to create a VirtualFile root.

The other two methods, on the other hand, use default settings for root creation. The difference between **`createNewRoot`** and **`getRoot`** is in caching details, which are covered later.

Example 8.4. Using `getVFS`

```
URL rootURL = ...; // get root url
VFS vfs = VFS.getVFS(rootURL);
// configure vfs instance
VirtualFile root1 = vfs.getRoot();
```

```
// or you can get root directly
VirtualFile root2 = VFS.crateNewRoot(rootURL);
VirtualFile root3 = VFS.getRoot(rootURL);
```

Another useful thing about VFS API is its implementation of a proper visitor pattern. It is very simple to recursively gather different resources, a task which is difficult to do with plain JDK resource loading.

Example 8.5. Recursively Gathering Resources

```
public interface VirtualFileVisitor {
    /**
     * Get the search attribues for this visitor
     *
     * @return the attributes
     */
    VisitorAttributes getAttributes();

    /**
     * Visit a virtual file
     *
     * @param virtualFile the virtual file being visited
     */
    void visit(VirtualFile virtualFile);
}

VirtualFile root = ...; // get root
VirtualFileVisitor visitor = new SuffixVisitor(".class"); // get all
classes
root.visit(visitor);
```

8.2. VFS ARCHITECTURE

While the public API is quite intuitive, real implementation details add complexity. Some concepts need to be explained in more detail.

Each time you create a VFS instance, its matching **`VFSContext`** instance is created. This creation is done via **`VFSContextFactory`**. Different protocols map to different **`VFSContextFactory`** instances. For example, **`file/vfsfile`** maps to **`FileSystemContextFactory`**, while **`zip/vfszip`** maps to **`ZipEntryContextFactory`**.

Each time a **`VirtualFile`** instance is created, its matching VirtualFileHandler is created. This VirtualFileHandler instance knows how to handle different resource types properly; the **`VirtualFile`** API only delegates invocations to its VirtualFileHandler reference.

The **`VFSContext`** instance knows how to create VirtualFileHandler instances accordingly to a resource type. For example, ZipEntryContextFactory creates **`ZipEntryContext`**, which then creates **`ZipEntryHandler`**.

8.3. EXISTING IMPLEMENTATIONS

Apart from files, directories (FileHandler) and zip archives (ZipEntryHandler) the Microcontainer also

supports other more advanced use cases. The first one is *Assembled*, which is similar to what Eclipse calls *Linked Resources*. Its purpose is to take existing resources from different trees, and "mock" them into single resource tree.

Example 8.6. Implementation of Assembled VirtualFileHandlers

```
AssembledDirectory sar =
AssembledContextFactory.getInstance().create("assembled.sar");

URL url = getResource("/vfs/test/jar1.jar");
VirtualFile jar1 = VFS.getRoot(url);
sar.addChild(jar1);

url = getResource("/tmp/app/ext.jar");
VirtualFile ext1 = VFS.getRoot(url);
sar.addChild(ext);

AssembledDirectory metainf = sar.mkdir("META-INF");

url = getResource("/config/jboss-service.xml");
VirtualFile serviceVF = VFS.getRoot(url);
metainf.addChild(serviceVF);

AssembledDirectory app = sar.mkdir("app.jar");
url = getResource("/app/someapp/classes");
VirtualFile appVF = VFS.getRoot(url);
app.addPath(appVF, new SuffixFilter(".class"));
```

Another implementation is *in-memory* files. This implementation arose out of a need to easily handle AOP generated bytes. Instead of using temporary files, you can drop bytes into in-memory VirtualFileHandlers.

Example 8.7. Implementation of In-Memory VirtualFileHandlers

```
URL url = new URL("vfsmemory://aopdomain/org/acme/test/Test.class");
byte[] bytes = ...; // some AOP generated class bytes
MemoryFileFactory.putFile(url,  bytes);

VirtualFile classFile = VFS.getVirtualFile(new
URL("vfsmemory://aopdomain"), "org/acme/test/Test.class");
InputStream bis = classFile.openStream(); // e.g. load class from input
stream
```

8.4. EXTENSION HOOKS

It is easy to extend VFS with a new protocol, similar to what we've done with **`Assembled`** and **`Memory`**. All you need is a combination of **`VFSContexFactory`**, **`VFSContext`**, **`VirtualFileHandler`**, **`FileHandlerPlugin`**, and **`URLStreamHandler`** implementations. The **`VFSContextFactory`** is trivial, while the others depend on the complexity of your task. You could implement **`rar`**, **`tar`**, **`gzip`**, or even **`remote`** access.

After implementing a new protocol, register the new VFSContextFactory with VFSContextFactoryLocator.

8.5. FEATURES

One of the first major problems the Microcontainer developers faced was proper usage of nested resources, more specifically nested jar files: For example, normal ear deployments: **`gema.ear/ui.war/WEB-INF/lib/struts.jar`**.

In order to read contents of **`struts.jar`** we have two options:

- handle resources in memory
- create top level temporary copies of nested jars, recursively

The first option is easier to implement, but it's very memory-consuming, requiring potentially large applications to reside in memory. The other approach leaves behind a large number of temporary files, which should be invisible to the end user and therefore should disappear after undeployment.

Consider the following scenario: A user accesses a VFS URL instance, which points to some nested resource.

The way plain VFS would handle this is to re-create the entire path from scratch: it would unpack nested resources over and over again. This leads to a great number of temporary files.

The Microcontainer avoids this by using **`VFSRegistry`**, **`VFSCache`**, and **`TempInfo`**.

When you ask for VirtualFile over VFS (**`getRoot`**, not **`createNewRoot`**), VFS asks the **`VFSRegistry`** implementation to provide the file. The existing **`DefaultVFSRegistry`** first checks if a matching root **`VFSContext`** exists for the provided URI. If it does, **`DefaultVFSRegistry`** first tries to navigate to the existing **`TempInfo`** (link to a temporary files), falling back to regular navigation if no such temporary file exists. In this way you completely reuse any temporary files which have already been unpacked, saving time and disk space. If no matching **`VFSContext`** is found in cache, the code will create a new **`VFSCache`** entry and continue with default navigation.

Determining how the **`VFSCache`** handles cached VFSContext entries depends on the implementation used. **`VFSCache`** is configurable via **`VFSCacheFactory`**. By default, nothing is cached, but there are a few useful existing **`VFSCache`** implementations, using algorithms such as **`Least Recently Used (LRU)`** or **`timed cache`**.

CHAPTER 9. THE CLASSLOADING LAYER

JBoss has always had a unique way of dealing with classloading, and the new classloading layer that comes with the Microcontainer is no exception ClassLoading is an optional add-on that you can use when you want non-default classloading. With the rising demand for OSGi-style classloading, and a number of new Java classloading specifications on the horizon, the changes to the ClassLoading layer of EAP 5.1 are useful and timely.

The Microcontainer ClassLoading layer is an abstraction layer. Most of the details are hidden behind private and package-private methods, without compromising the extensibility and functionality available through public classes and methods that make the API. This means that you code against policy and not against classloader details.

The ClassLoader project is split into 3 sub-projects

- classloader
- classloading
- classloading-vfs

`classloader` contains a custom **`java.lang.ClassLoader`** extension without any specific classloading policy. A classloading policy includes knowledge of where to load from and how to load.

`Classloading` is an extension of Microcontainer's dependency mechanisms. Its VFS-backed implementation is **`classloading-vfs`**. See Chapter 8, *The Virtual File System* for more information on VFS.

9.1. CLASSLOADER

The **`ClassLoader`** implementation supports pluggable policies and is itself a final class, not meant to be altered. To write your own ClassLoader implementations, write a **`ClassLoaderPolicy`** which provides a simpler API for locating classes and resources, and for specifying other rules associated with the classloader.

To customize classloading, instantiate a **`ClassLoaderPolicy`** and register it with a **`ClassLoaderSystem`** to create a custom **`ClassLoader`**. You can also create a **`ClassLoaderDomain`** to partition the **`ClassLoaderSystem`**.

The **`ClassLoader`** layer also includes the implementation of things like DelegateLoader model, classloading, resource filters, and parent-child delegation policies.

The run-time is JMX enabled to expose the policy used for each classloader. It also provides classloading statistics and debugging methods to help determine where things are loaded from.

Example 9.1. `ClassLoaderPolicy` Class

The **`ClassLoaderPolicy`** controls the way your classloading works.

```
public abstract class ClassLoaderPolicy extends BaseClassLoaderPolicy {
    public DelegateLoader getExported()

        public String[] getPackageNames()

        protected List<? extends DelegateLoader> getDelegates()
```

```
        protected boolean isImportAll()
        protected boolean isCacheable()
        protected boolean isBlackListable()

        public abstract URL getResource(String path);

    public InputStream getResourceAsStream(String path)

        public abstract void getResources(String name, Set<URL> urls)
throws IOException;

    protected ProtectionDomain getProtectionDomain(String className,
String path)
        public PackageInformation getPackageInformation(String
packageName)
        public PackageInformation getClassPackageInformation(String
className, String packageName)

        protected ClassLoader isJDKRequest(String name)
        }
}
```

The following two examples of **`ClassLoaderPolicy`**. The first one retrieves resources based on regular expressions, while the second one handles encrypted resources.

Example 9.2. ClassLoaderPolicy with Regular Expression Support

```
public class RegexpClassLoaderPolicy extends ClassLoaderPolicy {
    private VirtualFile[] roots;
    private String[] packageNames;

    public RegexpClassLoaderPolicy(VirtualFile[] roots)
    {
 this.roots = roots;
    }

    @Override
 public String[] getPackageNames()
    {
 if (packageNames == null)
     {
  Set<String> exportedPackages =
PackageVisitor.determineAllPackages(roots, null, ExportAll.NON_EMPTY,
null, null, null);
  packageNames = exportedPackages.toArray(new
String[exportedPackages.size()]);
     }
 return packageNames;
```

```
    }

    protected Pattern createPattern(String regexp)
    {
 boolean outside = true;
 StringBuilder builder = new StringBuilder();
 for (int i = 0; i < regexp.length(); i++)
     {
  char ch = regexp.charAt(i);
  if ((ch == '[' || ch == ']' || ch == '.') && escaped(regexp, i) ==
false)
      {
   switch (ch)
       {
       case '[' : outside = false; break;
       case ']' : outside = true; break;
       case '.' : if (outside) builder.append("\\"); break;
       }
      }

  builder.append(ch);
     }
 return Pattern.compile(builder.toString());
    }

    protected boolean escaped(String regexp, int i)
    {
 return i > 0 && regexp.charAt(i - 1) == '\\';
    }

    public URL getResource(String path)
    {
 Pattern pattern = createPattern(path);
 for (VirtualFile root : roots)
     {
  URL url = findURL(root, root, pattern);
  if (url != null)
      return url;
     }
 return null;
    }

    private URL findURL(VirtualFile root, VirtualFile file, Pattern
pattern)
    {
 try
     {
  String path = AbstractStructureDeployer.getRelativePath(root, file);
  Matcher matcher = pattern.matcher(path);
  if (matcher.matches())
      return file.toURL();

  List<VirtualFile> children = file.getChildren();
  for (VirtualFile child : children)
      {
   URL url = findURL(root, child, pattern);
```

```
    if (url != null)
        return url;
       }

   return null;
     }
  catch (Exception e)
     {
   throw new RuntimeException(e);
     }
    }

    public void getResources(String name, Set<URL> urls) throws
IOException
    {
  Pattern pattern = createPattern(name);
  for (VirtualFile root : roots)
     {
   RegexpVisitor visitor = new RegexpVisitor(root, pattern);
   root.visit(visitor);
   urls.addAll(visitor.getUrls());
     }
    }

    private static class RegexpVisitor implements VirtualFileVisitor
    {
  private VirtualFile root;
  private Pattern pattern;
  private Set<URL> urls = new HashSet<URL>();

  private RegexpVisitor(VirtualFile root, Pattern pattern)
  {
      this.root = root;
      this.pattern = pattern;
  }

  public VisitorAttributes getAttributes()
  {
      return VisitorAttributes.RECURSE_LEAVES_ONLY;
  }

  public void visit(VirtualFile file)
  {
      try
   {
       String path = AbstractStructureDeployer.getRelativePath(root,
file);
       Matcher matcher = pattern.matcher(path);
       if (matcher.matches())
    urls.add(file.toURL());
   }
      catch (Exception e)
   {
       throw new RuntimeException(e);
   }
  }
```

```
        public Set<URL> getUrls()
        {
            return urls;
        }
    }
}
```

RegexpClassLoaderPolicy uses a simplistic mechanism to find matching resources. Real-world implementations would be more comprehensive and elegant.

```
public class RegexpService extends PrintService {
    public void start() throws Exception
    {
 System.out.println();

 ClassLoader cl = getClass().getClassLoader();
 Enumeration<URL> urls = cl.getResources("config/[^.]+\\.[^.]{1,4}");
 while (urls.hasMoreElements())
     {
  URL url = urls.nextElement();
  print(url.openStream(), url.toExternalForm());
     }
    }
}
```

The regexp service uses the regular expression pattern **config/[^.]+\\.[^.]{1,4}** to list resources under the **config//** directory. The suffix length is limited, such that file names such as excluded.properties will be ignored.

Example 9.3. ClassLoaderPolicy with Encryption Support

```
public class CrypterClassLoaderPolicy extends VFSClassLoaderPolicy {
    private Crypter crypter;

    public CrypterClassLoaderPolicy(String name, VirtualFile[] roots,
VirtualFile[] excludedRoots, Crypter crypter) {
        super(name, roots, excludedRoots);
        this.crypter = crypter;
    }

    @Override
    public URL getResource(String path) {
        try
```

```
            {
                URL resource = super.getResource(path);
                return wrap(resource);
            }
        catch (IOException e)
            {
                throw new RuntimeException(e);
            }
    }

    @Override
    public InputStream getResourceAsStream(String path) {
        InputStream stream = super.getResourceAsStream(path);
        return crypter.crypt(stream);
    }

    @Override
    public void getResources(String name, Set<URL> urls) throws
IOException {
        super.getResources(name, urls);
        Set<URL> temp = new HashSet<URL>(urls.size());
        for (URL url : urls)
            {
                temp.add(wrap(url));
            }
        urls.clear();
        urls.addAll(temp);
    }

    protected URL wrap(URL url) throws IOException {
        return new URL(url.getProtocol(), url.getHost(), url.getPort(),
url.getFile(), new CrypterURLStreamHandler(crypter));
    }
}
```

Example 9.3, “ClassLoaderPolicy with Encryption Support” shows how to encrypt JARs. You can configure which resources to encrypt by specifying a proper filter. Here, everything is encrypted except for the contents of the **META-INF/** directory.

```
public class EncryptedService extends PrintService {
    public void start() throws Exception
    {
 ClassLoader cl = getClass().getClassLoader();

 URL url = cl.getResource("config/settings.txt");
 if (url == null)
     throw new IllegalArgumentException("No such settings.txt.");

 InputStream is = url.openStream();
 print(is, "Printing settings:\n");
```

```
    is = cl.getResourceAsStream("config/properties.xml");
    if (is == null)
        throw new IllegalArgumentException("No such properties.xml.");

    print(is, "\nPrinting properties:\n");
        }
    }
```

This service prints out the contents of two configuration files. It shows that decryption of any encrypted resources is hidden behind the classloading layer.

To properly test this, either encrypt the policy module yourself or use an existing encrypted one. To put this into action, you need to properly tie EncryptedService to **`ClassLoaderSystem`** and deployers.

Partitioning **`ClassLoaderSystem`** is discussed later in this chapter.

9.2. CLASSLOADING

Instead of using the **`ClassLoader`** abstraction directly, you can create **`ClassLoading`** modules which contain declarations of **`ClassLoader`** dependencies. Once the dependencies are specified the **`ClassLoaderPolicy`**s are constructed and wired together accordingly.

To facilitate defining the **`ClassLoaders`** before they actually exist, the abstraction includes a **`ClassLoadingMetaData`** model.

The **`ClassLoadingMetaData`** can be exposed as a Managed Object within the new JBoss EAP profile service. This helps system administrators to deal with more abstract policy details rather than the implementation details.

Example 9.4. ClassLoadingMetaData Exposed as a Managed Object

```
public class ClassLoadingMetaData extends NameAndVersionSupport {
    /** The serialVersionUID */
    private static final long serialVersionUID = -2782951093046585620L;

    /** The classloading domain */
    private String domain;

    /** The parent domain */
    private String parentDomain;

    /** Whether to make a subdeployment classloader a top-level
classloader */
    private boolean topLevelClassLoader = false;

    /** Whether to enforce j2se classloading compliance */
    private boolean j2seClassLoadingCompliance = true;
```

```
    /** Whether we are cacheable */
    private boolean cacheable = true;

    /** Whether we are blacklistable */
    private boolean blackListable = true;

    /** Whether to export all */
    private ExportAll exportAll;

    /** Whether to import all */
    private boolean importAll;

    /** The included packages */
    private String includedPackages;

    /** The excluded packages */
    private String excludedPackages;

    /** The excluded for export */
    private String excludedExportPackages;

    /** The included packages */
    private ClassFilter included;

    /** The excluded packages */
    private ClassFilter excluded;

    /** The excluded for export */
    private ClassFilter excludedExport;

    /** The requirements */
    private RequirementsMetaData requirements = new
RequirementsMetaData();

    /** The capabilities */
    private CapabilitiesMetaData capabilities = new
CapabilitiesMetaData();

    ... setters & getters
```

Example 9.5, “ClassLoading API Defined in XML” and Example 9.6, “ClassLoading API Defined in Java” show the ClassLoading API defined in XML and Java, respectively.

Example 9.5. ClassLoading API Defined in XML

```
<classloading xmlns="urn:jboss:classloading:1.0"
      name="ptd-jsf-1.0.war"
      domain="ptd-jsf-1.0.war"
      parent-domain="ptd-ear-1.0.ear"
```

```
        export-all="NON_EMPTY"
        import-all="true"
        parent-first="true"/>
```

Example 9.6. ClassLoading API Defined in Java

```
ClassLoadingMetaData clmd = new ClassLoadingMetaData();
if (name != null)
    clmd.setDomain(name + "_Domain");
clmd.setParentDomain(parentDomain);
clmd.setImportAll(true);
clmd.setExportAll(ExportAll.NON_EMPTY);
clmd.setVersion(Version.DEFAULT_VERSION);
```

You can add ClassLoadingMetaData to your deployment either programmatically, or declaratively, via **`jboss-classloading.xml`**.

Example 9.7. Adding ClassLoadingMetaData Using `jboss-classloading.xml`

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="DefaultDomain"
       top-level-classloader="true"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

The DefaultDomain is shared among all the applications that do not define their own domains.

Example 9.8. Typical Domain-Level Isolation

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="IsolatedDomain"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

Example 9.9. Isolation with a Specific Parent

```
<classloading xmlns="urn:jboss:classloading:1.0"
       domain="IsolatedWithParentDomain"
       parent-domain="DefaultDomain"
       export-all="NON_EMPTY"
       import-all="true">
</classloading>
```

Example 9.10. Non-Compliance with j2seClassLoadingCompliance

```
<classloading xmlns="urn:jboss:classloading:1.0"
       parent-first="false">
</classloading>
```

.war deployments use this method by default. Instead of doing default parent-first lookups, you first check your own resources.

Example 9.11. Typical OSGi Implementation

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <package name="org.jboss.dependency.spi"/>
  </requirements>
  <capabilities>
    <package name="org.jboss.cache.api"/>
    <package name="org.jboss.kernel.spi"/>
  </capabilities>
</classloading>
```

Example 9.12. Importing and Exporting Whole Modules and Libraries, Rather than Fine-Grained Packages

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <module name="jboss-reflect.jar"/>
  </requirements>
  <capabilities>
    <module name="jboss-cache.jar"/>
  </capabilities>
</classloading>
```

```
<classloading xmlns="urn:jboss:classloading:1.0">
  <requirements>
    <package name="si.acme.foobar"/>
    <module name="jboss-reflect.jar"/>
  </requirements>
  <capabilities>
    <package name="org.alesj.cl"/>
    <module name="jboss-cache.jar"/>
  </capabilities>
</classloading>
```

You can also mix the requirements and capabilities types, using packages and modules.

The classloading sub-project uses a very small resource-visitor-pattern implementation.

In the **ClassLoader** project, the connection between deployment and classloading is done through the **Module** class, which holds all of the required information to properly apply restrictions on the visitor pattern, such as filtering.

Example 9.13. The ResourceVisitor and ResourceContext Interfaces

```
public interface ResourceVisitor {
    ResourceFilter getFilter();

    void visit(ResourceContext resource);
}

public interface ResourceContext {
    URL getUrl();

    ClassLoader getClassLoader();

    String getResourceName();

    String getClassName();

    boolean isClass();

    Class<?> loadClass();

    InputStream getInputStream() throws IOException;

    byte[] getBytes() throws IOException;
```

```
}
```

To use the module, instantiate your ResourceVisitor instance and pass it to **`Module::visit`** method. This feature is used in the deployment framework to index annotations usage in deployments.

9.3. CLASSLOADING VFS

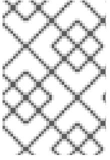
These examples provide an implementation of the **`ClassLoaderPolicy`** that uses a JBoss Virtual File System project to load classes and resources. You can use this idea directly or in combination with a classloading framework.

Optionally, you can set up your modules inside the Microcontainer configuration.

Example 9.14. Classloading Module Deployer

```
<deployment xmlns="urn:jboss:bean-deployer:2.0">

  <classloader name="anys-classloader"
xmlns="urn:jboss:classloader:1.0" import-all="true" domain="Anys"
parent-domain="DefaultDomain">
    <capabilities>
      <package
name="org.jboss.test.deployers.vfs.reflect.support.web"/>
    </capabilities>
    <root>${jboss.tests.url}</root>
  </classloader>

  <bean name="AnyServlet"
class="org.jboss.test.deployers.vfs.reflect.support.web.AnyServlet">
    <classloader><inject bean="anys-classloader:0.0.0"/></classloader>
  </bean>

</deployment>
```

The **`VFSClassLoaderFactory`** class transforms the XML deployer into a **`VFSClassLoaderPolicyModule`**, which then creates the actual **`ClassLoader`** instance. You can then directly use this new **`ClassLoader`** instance with your beans.



NOTE

`VFSClassLoaderFactory` extends **`ClassLoadingMetaData`**, so all examples pertaining to **`ClassLoadingMetaData`** apply in this case as well.

CHAPTER 10. THE VIRTUAL DEPLOYMENT FRAMEWORK

The new *Virtual Deployment Framework (VDF)* is an improved way to manage deployers in the Microcontainer. This chapter details some of its useful features.

10.1. AGNOSTIC HANDLING OF DEPLOYMENT TYPES

The traditional type of virtual deployment is based on classes which already exist in a shared class-space or domain. In this case, the end product is a new service installed onto the server from your main client. The traditional way to do this is to upload a descriptor file. The new VDF simplifies this process by passing over the bytes and serializing them into a **`Deployment`** instance.

The other type of deployment, which extends the first one, is a plain file-system-based deployment, backed up by Microcontainer VFS. This approach is described in more detail in Chapter 8, *The Virtual File System*.

10.2. SEPARATION OF STRUCTURE RECOGNITION FROM DEPLOYMENT LIFECYCLE LOGIC

In order to do any real work on top of a deployment, you must first understand its structure, including its classpaths and metadata locations.

Metadata locations include the configuration files such as **`my-jboss-beans.xml`**, **`web.xml`**, **`ejb-jar.xml`**. Classpaths are classloader roots, such as **`WEB-INF/classes`** or **`myapp.ear/lib`**.

Bearing the structure in mind, you can proceed to actual deployment handling.

A Typical Deployment Lifecycle

1. The **`MainDeployer`** passes the deployment to the set of **`StructuralDeployer`**s for recognition, and receives back a Deployment context.

2. Next, the **`MainDeployer`** passes the resulting Deployment context to the **`Deployers`** for handling by the appropriate **`Deployer`**.

In this way, the MainDeployer is a broker with the responsibility of deciding which Deployers to use.

In the case of virtual or programmatic deployment, an existing predetermined StructureMetaData information reads the structure information and handles it in one of the ways explained in Handling StructuredMetaData Information.

Handling StructuredMetaData Information

VFS-based deployments

the structure recognition is forwarded to a set of StructureDeployers.

JEE-specification-defined structures

we have matching StructureDeployer implementations:

- EarStructure
- WarStructure

- JarStructure

DeclarativeStructures

looks for **META-INF/jboss-structure.xml** file inside your deployment, and parses it to construct a proper **StructureMetaData**.

FileStructures

only recognizes known configuration files, such as files like **-jboss-beans.xml** or **-service.xml**.

Example 10.1. An example of jboss-structure.xml

```
<structure>
  <context
comparator="org.jboss.test.deployment.test.SomeDeploymentComparatorTo
p">
    <path name=""/>
    <metaDataPath>
      <path name="META-INF"/>
    </metaDataPath>
    <classpath>
      <path name="lib" suffixes=".jar"/>
    </classpath>
  </context>
</structure>
```

In the case of EarStructure, first recognize a top level deployment, then recursively process sub-deployments.

You can implement a custom **StructureDeployer** with the help of the generic **GroupingStructure** class provided by the generic **StructureDeployer** interface.

After you have a recognized deployment structure, you can pass it to real deployers. The Deployers object knows how to deal with the real deployers, using a chain of deployers per **DeploymentStage**.

Example 10.2. Deployment Stages

```
public interface DeploymentStages {
   /** The not installed stage - nothing is done here */
   DeploymentStage NOT_INSTALLED = new DeploymentStage("Not Installed");

   /** The pre parse stage - where pre parsing stuff can be prepared;
altDD, ignore, ... */
   DeploymentStage PRE_PARSE = new DeploymentStage("PreParse",
NOT_INSTALLED);

   /** The parse stage - where metadata is read */
   DeploymentStage PARSE = new DeploymentStage("Parse", PRE_PARSE);

   /** The post parse stage - where metadata can be fixed up */
   DeploymentStage POST_PARSE = new DeploymentStage("PostParse",
```

```
PARSE);

   /** The pre describe stage - where default dependencies metadata can
be created */
   DeploymentStage PRE_DESCRIBE = new DeploymentStage("PreDescribe",
POST_PARSE);

   /** The describe stage - where dependencies are established */
   DeploymentStage DESCRIBE = new DeploymentStage("Describe",
PRE_DESCRIBE);

   /** The classloader stage - where classloaders are created */
   DeploymentStage CLASSLOADER = new DeploymentStage("ClassLoader",
DESCRIBE);

   /** The post classloader stage - e.g. aop */
   DeploymentStage POST_CLASSLOADER = new
DeploymentStage("PostClassLoader", CLASSLOADER);

   /** The pre real stage - where before real deployments are done */
   DeploymentStage PRE_REAL = new DeploymentStage("PreReal",
POST_CLASSLOADER);

   /** The real stage - where real deployment processing is done */
   DeploymentStage REAL = new DeploymentStage("Real", PRE_REAL);

   /** The installed stage - could be used to provide valve in future?
*/
   DeploymentStage INSTALLED = new DeploymentStage("Installed", REAL);
}
```

Preexisting deployment stages are mapped to the Microcontainer's built-in controller states. They provide a deployment-lifecycle-centric view of generic controller states.

Inside Deployers, the deployment is converted into the Microcontainer's component **DeploymentControllerContext**. The Microcontainer's state machine handles dependencies.

Deployments are handled sequentially by deployment stage. For each deployer, the entire deployed hierarchy order is handled, using the deployer's **parent-first** property. This property is set to **true** by default.

You can also specify which hierarchy levels your deployer handles. You can choose **all**, **top level**, **components only**, or **no components**.

The way the Microcontainer handles component models and dependency handling applies here as well. If there are unresolved dependencies, the deployment will wait in the current state, potentially reporting an error if the current state is not the required state.

Adding a new deployer is accomplished by extending one of the many existing helper deployers.

Some deployers actually need VFS backed deployment, while others can use a general deployment. In most cases the parsing deployers are the ones that need VFS backing.

WARNING

Also be aware that deployers run recursively through every deployment, sub-deployment, and component. Your code needs to determine, as early in the process as possible, whether the deployer should handle the current deployment or not.

Example 10.3. Simple Deployer which Outputs Information About Its Deployment

```
public class StdioDeployer extends AbstractDeployer {
   public void deploy(DeploymentUnit unit) throws DeploymentException
   {
      System.out.println("Deploying unit: " + unit);
   }

   @Override
   public void undeploy(DeploymentUnit unit)
   {
      System.out.println("Undeploying unit: " + unit);
   }
}
```

Add this description into one of the **-jboss-beans.xml** files in **deployers/** directory in JBoss Application Server, and **MainDeployerImpl** bean will pick up this deployer via the Microcontainer's IoC callback handling.

Example 10.4. Simple Deployment Descriptor

```
<bean name="StdioDeployer" class="org.jboss.acme.StdioDeployer"/>
```

10.3. NATURAL FLOW CONTROL IN THE FORM OF ATTACHMENTS

VDF includes a mechanism called *attachments*, which facilitates the passing of information from one deployer to the next. Attachments are implemented as a slightly-enhanced **java.util.Map**, whose entries each represent an attachment.

Some deployers are producers, while others are consumers. The same deployer can also perform both functions. Some deployers create metadata or utility instances, putting them into the *attachments* map. Other deployers only declare their need for these attachments and pull the data from the attachments map, before doing additional work on that data.

Natural order refers to the way that deployers are ordered. A common natural order uses the relative terms *before* and *after*. However, with the attachments mechanism already in place, you can order deployers by the way in which they produce and/or consume the attachments.

Each attachment has a key, and deployers pass keys to the attachments they produce. If the deployer produces an attachment, then that key is called *output*. If the deployer consumes an attachment, then that key is called *input*.

Deployers have *ordinary* inputs and *required* inputs. Ordinary inputs are only used to help determine the natural order. Required inputs also help determine order, but they have another function too. They help to determine if the deployer is actually relevant for the given deployment, by checking to see if an attachment corresponding to that required input exists in the attachments map.

WARNING

While relative ordering is still supported, it is considered bad practice, and could be desupported in future releases.

10.4. CLIENT, USER, AND SERVER USAGE AND IMPLEMENTATION DETAILS

These features hide the implementation details, making the usage less error-prone, while at the same time streamlining the development process.

The goal is for clients to only see a Deployment API, while developers see a DeploymentUnit, and server implementation details are contained in a DeploymentContext. Only the necessary information is exposed to a particular level of deployment's lifecycle.

Components have already been mentioned as part of deployers' hierarchy handling. While top level deployment and sub-deployments are a natural representation of the deployment's structure hierarchy, components are a new VDF concept. The idea of components is that they have a 1:1 mapping with the **`ControllerContexts`** inside the Microcontainer. See Why Components Map 1:1 with the **`ControllerContexts`** for the reasons behind this assertion.

Why Components Map 1:1 with the `ControllerContexts`

Naming

The component unit's name is the same as the **`ControllerContext`**'s name.

get*Scope() and get*MetaData()

return the same MDR context that will be used by the Microcontainer for that instance.

IncompleteDeploymentException (IDE)

In order for the IDE to print out what dependencies are missing for a deployment, it needs to know the ControllerContext names. It discovers the name by collecting the Component DeploymentUnit's names in Component Deployers that specify them, such as **`BeanMetaDataDeployer`** or the **`setUseUnitName()`** method in **`AbstractRealDeployer`**.

10.5. SINGLE STATE MACHINE

All Microcontainer components are handled by a single entry point, or single state machine. Deployments are no exception.

You can take advantage of this feature by using the **jboss-dependency.xml** configuration file in your deployments.

Example 10.5. jboss-dependency.xml

```
<dependency xmlns="urn:jboss:dependency:1.0">
  <item whenRequired="Real"
dependentState="Create">TransactionManager</item> (1)
  <item>my-human-readable-deployment-alias</item> (2)
</dependency>
```

Note the artificial call-outs in the XML: (1) and (2).

(1) shows how to describe dependency on another service. This example requires **TransactionManager** to be created before the deployment is in the 'Real' stage.

(2) is a bit more complex, since you are missing additional information. By default, deployment names inside the Microcontainer are URI names, which makes typing them by hand an error prone proposition. So, in order to be able to easily declare dependence on other deployments, you need an aliasing mechanism to avoid URI names. You can add a plain text file named **aliases.txt** into your deployment. Each line of the file contains an alias, giving a deployment archive one or more simple names used to refer to it.

10.6. SCANNING CLASSES FOR ANNOTATIONS

Current JEE specifications reduce the number of configuration files, but the container is now required to do most of the work using @annotations. In order to get @annotation info, containers must scan classes. This scanning creates a performance penalty.

But to reduce the amount of scanning, the Microcontainer provides another descriptor hook, by means of **jboss-scanning.xml**.

Example 10.6. jboss-scanning.xml

```
<scanning xmlns="urn:jboss:scanning:1.0">
  <path name="myejbs.jar">
    <include name="com.acme.foo"/>
    <exclude name="com.acme.foo.bar"/>
  </path>
  <path name="my.war/WEB-INF/classes">
    <include name="com.acme.foo"/>
  </path>
</scanning>
```

This example shows a simple description of relative paths to include or exclude when scanning for Java Enterprise Edition version 5 and greater annotated metadata information.

APPENDIX A. REVISION HISTORY

Revision 5.1.0-111.400	**2013-10-31**	**Rüdiger Landmann**

Rebuild with publican 4.0.0

Revision 5.1.0-111	**2012-07-18**	**Anthony Towns**

Rebuild for Publican 3.0

Revision 5.1.0-100	**Wed Sep 15 2010**	**Misty Stanley-Jones**

JBPAPP-5076 - Fix mis-matches between examples and their text

Changed version number in line with new versioning requirements.

Revised for JBoss Enterprise Application Platform 5.1.0.GA.