Getting Started with the JBoss Microcontainer

A Guide for POJO Developers

Getting Started with the JBoss Microcontainer: A Guide for POJO Developers

2.0.0

Table of Contents

List of Figures

List of Examples

Target Audience

This guide is aimed at developers who want to use the Microcontainer to assemble their own applications or use it to develop shared services in the JBoss Application Server.

Preface

Commercial development support, production support and training for the JBoss Microcontainer is available through JBoss Inc. [http://www.jboss.com] The JBoss Microcontainer is a project of the JEMS product suite.

Authors:

- Adrian Brock - JBoss Microcontainer Project Lead and Chief Scientist of JBoss Inc.

  Scott Stark - VP Architecture & Technology

  Kabir Khan - JBoss AOP Project Lead

  Ales Justin - JBoss Core Developer

Chapter 1. Introduction to the JBoss Microcontainer

The JBoss Microcontainer provides an environment to configure and manage POJOs (plain old java objects). It is designed to reproduce the existing JBoss JMX Microkernel but targeted at POJO environments. As such it can be used standalone outside the JBoss Application Server.

As we mention JBoss Application Server, while rewriting its Microkernel to a POJO environment, there were other aspects that also needed to be considered while changing the core functionality. New modules or subprojects were introduced to the Microcontainer project to tightly integrate those aspects with the new POJO approach and to eventually solve some of the basic problems that arose over the years with the changes application server went through. We will discuss these modules and subproject later on.

At its core, the JBoss Microcontainer is a generic "dependency injection" framework. Its primary function is to instantiate objects, figure out their dependencies (e.g., object A must be instantiated before object B), and then manage the relationship between those objects (e.g., object A is a property of object B). With the JBoss Microcontainer, you can build applications or shared services using simple POJOs. The JBoss Microcontainer is responsible for assembling the POJOs together according to an externally defined XML configuration file. It decouples the components in an application and makes the application easy to unit-test.

The JBoss Microcontainer is a big object factory that manages objects. In this sense, it is similar to existing dependency injection frameworks such as the Spring framework and HiveMind framework. However, JBoss Microcontainer also has several important new features that sets it apart from existing frameworks.

- JBoss Microcontainer supports life cycles for POJO components. It gives you fine-grained control over exactly how the objects are created, instantiated, and destroyed.
- JBoss Microcontainer manages dependencies between objects. For instance, you can declare that object A must be instantiated before object B can be created.
- JBoss Microcontainer integrates with the JBoss AOP (Aspect Oriented Programming) framework. In the JBoss Microcontainer configuration file, you can easily wire aspects as services to POJOs.
- JBoss Microcontainer is used as a replacement for the JMX-based Microkernel in JBoss AS 5.x and above. All the existing features of MBeans are thightly integrated into new Microcontainer based ServiceController.
- JBoss Microcontainer is embedded in JBoss AS 4.x. It is the ideal choice if you need to develop shared services in JBoss AS.

Of course, you can also run JBoss Microcontainer outside of the JBoss AS. For instance, you can run it in a Java SE (e.g., Swing) application or in the Tomcat servlet container. So, JBoss Microcontainer primarily targets three types of developers.

- Framework developers can use JBoss Microcontainer to assemble custom server frameworks. For instance, the JBoss embeddable EJB3 framework is based on the Microcontainer.
- JBoss application developers can develop POJO services that nicely integrate into the server and can be shared across applications.
- Application developers can write lightweight applications that makes use of services from a variety of sources (e.g., transaction service from JBoss, persistence service from Hibernate, and HTTP service from Tomcat).

A typical JBoss Microcontainer application or service include a set of POJOs that complete business tasks, as well as an XML configuration file called `META-INF/jboss-beans.xml` on the class path. The `jboss-beans.xml` file tells the JBoss Microcontianer how to assemble those POJOs together. For deployment, you can JAR the POJO classes, as well as the `META-INF/jboss-beans.xml` file together in a simple .jar archive file (see Chapter 5, *Packaging*).

This document takes you through some example deployments into JBoss Microcontainer 2.0 explaining how to configure POJOs and link them together through injection. We will also explain how to use other Microcontainer modules inside JBoss AS 5.x, especially the re-written deployers and classloader modules. Later, we will discuss how to do the same thing outside the application server.

Chapter 2. Download and Installation

First you need to download JBoss Microcontainer release from http://labs.jboss.com/jbossmc/downloads [http://labs.jboss.com/jbossmc/downloads]

Unpack the archive which will give you a microcontainer-x.y.z directory with the following subfolders:

- docs/api - javadocs for the Microcontainer
- docs/gettingstarted - this getting started documentation
- docs/licences - the licenses for the software
- examples - the examples explained in the next chapter
- lib - the libraries required to run the Microcontainer

You will also need a JDK of version 1.4.x+ and a copy of Apache Ant 1.6+

If you want to run the examples inside JBoss AS, you will need to download JBoss AS 5.x http://labs.jboss.com/jbossas/downloads [http://labs.jboss.com/jbossas/downloads]

Chapter 3. Glossary

A glossary to start

The glossary ...

Chapter 4. Starting Examples

A Simple IoC Examples

The best way to learn the Microcontainer is through examples. The Microcontainer distribution is bundled with several examples, which we will discuss in later this guide. In this section, let us first have a look at the `simple` example. It shows the structure of a simple Microcontainer application and how to run the application in both standalone and JBoss AS environments.

The `simple` example is located in the `examples/simple` directory of the Microcontainer distribution. It contains a single class under the `src/main` directory.

```
public class SimpleBean
{
   public SimpleBean()
   {
      System.out.println("SimpleBean() constructor");
   }
}
```

The `SimpleBean` object prints to the system console when it is instantiated via the default constructor. Now, we need to use the Microcontainer to instantiate a `SimpleBean` POJO. We do this by invoking the Microcontainer with the `src/resources/META-INF/jboss-beans.xml` configuration file.

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

   <bean name="Simple"
      class="org.jboss.example.microcontainer.simple.SimpleBean"/>

</deployment>
```

This configuration file tells the Microcontainer to create an instance of the `SimpleBean` POJO and manage it under the name `Simple` . When other objects in the application need to access this `SimpleBean` instance, they can simply ask the Microcontainer API for the `Simple` object. Essentially, we just created a `SimpleBean` singleton instance here. When we run this application, we are expected to see the `"SimpleBean() constructor"` printout on the console when the Microcontainer creates the `Simple` object.

Build and Package the Application

To build, package, and run the `simple` application, we can simply execute the `ant` command in the `examples/simple` directory. It runs the `build.xml` script, which further invokes the `examples/build-common.xml` script. The `compile` and `dist` tasks in the build script builds the application.

```
<target name="compile">

   <mkdir dir="build"/>

   <javac destdir="build"
      debug="on"
      deprecation="on"
      optimize="on"
      classpathref="compile.classpath">

      <src path="src"/>
   </javac>

</target>

<target name="dist" depends="compile">

   <copy todir="build">
      <fileset dir="src/resources"/>
   </copy>

   <mkdir dir="dist"/>

   <jar jarfile="dist/${ant.project.name}.beans" basedir="build"/>

</target>
```

The `compile` target compiles the Java source files into class files under the `build` directory. Then, the `dist` target packages the class files and the `META-INF/jboss-beans.xml` file together in a JAR file named `example-simple.jar` in the `dist` directory.

In the next two sections, let's discuss how to run the newly created application both as a standalone Java SE application and as a service in the JBoss AS.

This just skims the surface of the Microcontainer, showing the most common usecases. Other more complicated examples can be found in the tests (available in the source code repository).

Chapter 5. Packaging

We had a brief discussion on the `jboss-beans.xml` file in Chapter 4, *Starting Examples* . In this chapter, we will go into more depth on the packaging topic. As we will see later in Chapter 16, *Standalone* , the packaging is more of a convention rather than a requirement. The convention is recommended since it allows "deployments" to be used both standalone and inside JBoss AS.

The basic structure of Microcontainer deployment is a plain .jar file (see below). The jar archive can also be unpacked in a directory structure that looks the jar file. It contains a `META-INF/jboss-beans.xml` to describe what you want it to do. The contents of this xml file are described in ??? . Finally, the archive contains the classes and any resources just like any other jar file.

```
example.jar/
example.jar/META-INF/jboss-beans.xml
example.jar/com/acme/SomeClass.class
```

If you want to include a .jar file in an `.ear` deployment, you will need to reference in `META-INF/jboss-app.xml` .

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

   <!DOCTYPE jboss-app
      PUBLIC "-//JBoss//DTD J2EE Application 1.4//EN"
      "http://www.jboss.org/j2ee/dtd/jboss-app_4_0.dtd">

   <jboss-app>

      <module>
         <service>example.jar</service>
      </module>

   </jboss-app>
```

Chapter 6. Microcontainer core

Core ...

Basic Configuration and Usage

The Microcontainer's main purpose is to allow external configuration of POJOs. As we have seen in Chapter 4, *Starting Examples* , the POJOs in a Microcontainer applications are nothing special. The key element that drives the application is the `jboss-beans.xml` configuration file. So, in this chapter, we will look at the some of the common configurations in `jboss-beans.xml` .

Deployment

The `deployment` element acts as a container for many beans that are deployed together.

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

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

<!-- bean part of the deployment -->
<bean .../>

<!-- bean part of the deployment -->
<bean .../>

</deployment&gt;
```

Bean

The `bean` element is the main deployment component. It describes a single managed object in the runtime.

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

<bean name="Simple"
   class="org.jboss.example.microcontainer.simple.SimpleBean"/>
```

```
</deployment>
```

The example above from the `simple` example uses the default constructor of `SimpleBean` to create a new POJO.

```
new org.jboss.example.microcontainer.simple.SimpleBean();
```

It is given the name `"Simple"` such that it can be referenced elsewhere.

Construction

The example above uses the default constructor. What if you don't want to use some other constructor. The simplest mechanism just matches the number of parameters in the constructor. The example below is from the `constructor` example.

```
public class ConstructorBean
{
   ... ...

   public ConstructorBean(int integer)
   {
      ... ...
   }
}
```

The `jboss-beans.xml` element for creating the POJO using the above constructor is as follows.

```
<bean name="Integer"
   class="org.jboss.example.microcontainer.constructor.ConstructorBean">

   <constructor>
      <parameter>4</parameter>
   </constructor>
</bean>
```

The microcontainer would just use the following to create the `Integer` named component.

```
new ConstructorBean (4);
```

Sometimes there are two constructors with the same number of parameters. In this case, you must specify the types to resolve the ambiguity.

```
public class ConstructorBean {
public ConstructorBean(String string, int integer) {}
public ConstructorBean(String string, long long) {}
}
```

The following configuration creates a managed bean instance named `StringLong` using the second constructor listed above.

```
<bean name="StringLong"
   class="org.jboss.example.microcontainer.constructor.ConstructorBean">

   <constructor>
      <parameter>a string</parameter>
      <parameter class="long">10</parameter>
   </constructor>
</bean>
```

Behind the scene, the Micorcontainer invokes:

```
new ConstructorBean ("a string", (long) 10);
```

Note that you only have to be explicit enough to resolve the ambiguity.

Factories

Not all classes have pubic constructors. It is often good practice to use factories when you to have a choice of implementation for an interface. The microcontainer includes support for the several different types of factory. The simplest case is a static factory class with a static factory method like the following from the `factory` example.

```
public class StaticFactory
{
   public static FactoryCreatedBean createBean()
   {
      return new FactoryCreatedBean("StaticFactory.createBean()");
```

```
   }
}
```

The bean configuration tells the microcontainer to the use the `StaticFactory.createBean()` static method to create an instance of `FactoryCreatedBean` .

```
<bean name="StaticFactoryCreatedBean"
   class="org.jboss.example.microcontainer.factory.FactoryCreatedBean">

   <constructor factoryMethod="createBean"
      factoryClass="org.jboss.example.microcontainer.factory.StaticFactory"/>

</bean>
```

Of course, the factory class itself does not have to be static. The microcontainer can create a non-static factory itself as a managed object, and then use this factory object to create other bean objects. For instance, the `factory` example contains a singleton factory class example.

```
public class SingletonFactory
{
   private static SingletonFactory singleton;

   public synchronized static SingletonFactory getInstance()
   {
      if (singleton == null)
         singleton = new SingletonFactory();

      return singleton;
   }

   public FactoryCreatedBean createBean()
   {
      return new FactoryCreatedBean("SingletonFactory.createBean()");
   }

   private SingletonFactory()
   {
      System.out.println("SingletonFactory()");
   }
}
```

In the following configuration, we first create an instance of the `SingletonFactory` class under the name `"SingletonFactory"` using the `SingletonFactory.getInstance()` static method. Then, we use this factory object to create a `FactoryCreatedBean` instance under the name `SingletonFactoryCreatedBean` .

```
<bean name="SingletonFactory"
   class="org.jboss.example.microcontainer.factory.SingletonFactory">

   <constructor factoryMethod="getInstance"
      factoryClass="org.jboss.example.microcontainer.factory.SingletonFactory"/

</bean>

<bean name="SingletonFactoryCreatedBean"
   class="org.jboss.example.microcontainer.factory.FactoryCreatedBean">

   <constructor factoryMethod="createBean">
      <factory bean="SingletonFactory"/>
   </constructor>

</bean>
```

Like the constructor method we mentioned before, the factory method can also take call parameters via the nested `parameter` element.

Properties

It is possible to create all objects using factories and constructors, however many people use the JavaBean or MBean convention where an object is constructed using a default constructor and then configured using properties or attributes (i.e., setter methods). The following class from the `properties` example is a JavaBean style POJO.

```
public class PropertiesBean
{
   public PropertiesBean()
   {
      System.out.println("PropertiesBean()");
   }

   public void setTitle(String title)
   {
      System.out.println("setTitle: " + title);
   }
```

```
   public void setSubTitle(String subTitle)
   {
      System.out.println("setSubTitle: " + subTitle);
   }

   public void setLink(URL url)
   {
      System.out.println("setLink: " + url);
   }

   public void setNumber(Number number)
   {
      System.out.println("setNumber: " + number + " type=" + number.getClass().
   }
}
```

The configuration creates the `PropertiesBean` instance and then use the setter methods to set its properties.

```
<bean name="PropertiesBean"
   class="org.jboss.example.microcontainer.properties.PropertiesBean">

   <property name="title">JBoss Microcontainer property example</property>
   <property name="subTitle"><null/></property>
   <property name="link">http://www.jboss.org</property>
   <property name="number" class="java.lang.Long">4</property>
</bean>
```

Here we use the string representation of various objects, including the `null` value. They will be discussed in more detail in the next section.

String Values

Now, you probably noticed that we use string values to represent all kinds of objects in the `parameter` element for the constructors or factory methods, and in the `property` element for JavaBean properties.

In the most generic case, a JavaBean PropertyEditor [http://java.sun.com/j2se/1.4.2/docs/api/java/beans/PropertyEditor.html] can be used to convert a string to a specific type. JBoss already provides a large number of property editors for standard types. Please see the example below for the POJO class, the configuration, and the resultant Microcontainer action.

```
import java.beans.PropertyEditorSupport;

public class URLEditor extends PropertyEditorSupport
{
   public void setAsText(final String text)
   {
      setValue(new URL(text));
   }
}

public class Example
{
   public URL getLink() {}

   public void setLink(URL url) {}

}

<bean name="Name1" class=com.acme.Example">
   <property name="link">http://acme.com/index.html</property>
</bean>

Example example = new com.acme.Example();
PropertyEditor editor = PropertyEditorManager.findEditor(URL.class);
editor.setAsText("http://acme.com/index.html");
example.setLink((URL)editor.getValue());
```

Often the property takes an interface or abstract class, but you want to pass a specific implementation or a subclass. That is easy. Just specify the implementation class in the `property.class` attribute as show below.

```
public class Example
{
      public Number getNumber() {}

      public void setNumber(Number number) {}

}

<bean name="Name1" class=com.acme.Example">
   <property name="number" class="java.lang.Long">4</property>
</bean>

Example example = new com.acme.Example();
example.setNumber(new Long(4));
```

There is also a more long-winded form of value that we will see later when configuring collections.

```
public class Example
{
   public Number getNumber() {}

   public void setNumber(Number number) {}

}

<bean name="Name1" class=com.acme.Example">
   <property name="number">
      <value class="java.lang.Long">4</value>
   </property>
</bean>

Example example = new com.acme.Example();
example.setNumber(new Long(4));
```

Finally, the `null` value is trivial, `<null/>`. But, it needs to differentiated from the string "null". Please see the example below for the usage.

```
public class Example
{
      public String getTitle() {}

      public void setTitle(String string) {}

}

<!-- Wrong -->
<bean name="Name1" class=com.acme.Example">
   <property name="title">null</property>
</bean>

Example example = new com.acme.Example();
example.setTitle(new String("null"));

<!-- Correct -->
<bean name="Name1" class=com.acme.Example">
   <property name="title"><null/></property>
</bean>

Example example = new com.acme.Example();
example.setTitle(null);
```

Sometimes it is still usefull to be able to set or inject a value which can be morphed / progressed into needed type althogh existing type is not compatible (normally getting `ClassCastException`). This is mostly true for `java.lang.Number` subclasses. Let's see how `progression` is used on the following example (here we use `<inject />`, which is actually explained in the next chapter).

```
<bean name="FromBean" class="org.jboss.test.kernel.config.support.SimpleBean">
   <property name="adouble">123.456</property>
   <property name="AFloat" class="java.lang.Float">987.6543</property>
   <property name="anInt">314159</property>
</bean>

<bean name="SimpleBean" class="org.jboss.test.kernel.config.support.SimpleBean"
   <property name="anint">
      <inject bean="FromBean" property="adouble" />
   </property>

   <property name="AShort">
      <inject bean="FromBean" property="AFloat" />
   </property>

   <property name="AFloat">
      <inject bean="FromBean" property="anInt" />
   </property>
</bean>
```

SimpleBean is a plain JavaBean, property names reflect the field type: having a`double` property name means that a field is of type double. A`Float` corresponds to Float type. Etc.

You can change the progression behavior by setting System property `org.jboss.reflect.plugins.progressionConvertor`, putting the ProgressionConvertor's implementation fully qualified class name as value. By default SimpleProgressionConvertor implementation is used. Another existing implementation is NullProgressionConvertor, which doesn't do any actual progression.

Injections

Objects by themself are not very useful. They need to be linked together to form more complicated data structures. We have already seen one form of an injection when using factory instances above. Injections can be used anywhere a string value is used. All the examples that we have previously seen with strings could also be done with injections.

The `injection` example shows how dependency injection works in JBoss Microcontainer. The `InjectionBean` class has a `host` property, which is the `java.net.URL` type. We will inject an URL object into the bean via the microcontainer.

```
public class InjectionBean
{
   String name;

   public InjectionBean(String name)
   {
      this.name = name;
      System.out.println("InjectionBean() " + this);
   }

   public String toString()
   {
      return name;
   }

   public void setHost(String host)
   {
      System.out.println("setHost: " + host + " on " + this);
   }

   public void setOther(InjectionBean other)
   {
      System.out.println("setOther: " + other + " on " + this);
   }
}
```

The microcontainer creates the `URL` object first, and then passes the `URL` object as a property into the `InjectionBean1` object when it is instantiated.

```
<bean name="URL" class="java.net.URL">
   <constructor>
      <parameter>http://www.jboss.org/index.html</parameter>
   </constructor>
</bean>

<bean name="InjectionBean1"
   class="org.jboss.example.microcontainer.injection.InjectionBean">

   <constructor>
      <parameter>InjectionBean1</parameter>
   </constructor>

   <property name="host">
      <inject bean="URL" property="host"/>
   </property>
```

```
</bean>
```

The order of the `bean` elements does not matter. The microcontainer orders the beans into the correct order. For instance, in the above example, the `URL` object is always created before the `InjectionBean1` object since the latter is dependent on the former. But that leaves the problem of how to resolve circular dependencies. These can be resolved by specifying when you want the injection to occur. In the example below we say once `Circular1` is "Instantiated" (constructed) it is ok to configure it on `Circular2` . Normally, injections wait for the referenced bean to reach the state "Installed" (see later on life cycles).

```
<bean name="Circular1"
   class="org.jboss.example.microcontainer.injection.InjectionBean">

   <constructor>
      <parameter>Circular1</parameter>
   </constructor>

   <property name="other">
      <inject bean="Circular2"/>
   </property>

</bean>

<bean name="Circular2"
   class="org.jboss.example.microcontainer.injection.InjectionBean">

   <constructor>
      <parameter>Circular2</parameter>
   </constructor>

   <property name="other">
      <inject bean="Circular1" state="Instantiated"/>
   </property>
</bean>
```

Here is the order the microcontainer instantiates those objects.

```
InjectionBean Circular1 = new InjectionBean ();
InjectionBean Circular2 = new InjectionBean ();
Circular1.setOther(Circular2); // don't wait for a fully configured Circular1
Circular2.setOther(Circular1); // Complete the confguration of Circular2
```

Value Factory

Similar to using `inject`'s property attribute, we sometimes want to use other beans to create new values from bean's multi parameter methods.

```
<bean name="PropHolder" class="org.jboss.test.kernel.config.support.PropHolder"
   <property name="value">
      <value-factory bean="ldap" method="getValue">
         <parameter>xyz.key</parameter>
         <parameter>xyz</parameter>
         <parameter>
            <bean name="t" class="org.jboss.test.kernel.config.support.TrimTran
         </parameter>
      </value-factory>
   </property>
</bean>

<bean name="ldap" class="org.jboss.test.kernel.config.support.LDAPFactory">
   <constructor>
      <parameter>
         <map keyClass="java.lang.String" valueClass="java.lang.String">

            <entry>
               <key>foo.bar.key</key>
               <value>QWERT</value>
            </entry>

            <entry>
               <key>xyz.key</key>
               <value>QWERT</value>
            </entry>

         </map>

      </parameter>
   </constructor>
</bean>
```

For the quick usage there is also shorthand version of the value-factory element, having single parameter as string attribute.

```
<bean name="PropHolder" class="org.jboss.test.kernel.config.support.PropHolder"
   <constructor>
```

```
      <parameter>
         <value-factory bean="ldap" method="getValue" parameter="foo.bar.key"/>
      </parameter>
   </constructor>
</bean>
```

You can also define a default value for the value-factory return value.

```
<bean name="PropHolder" class="org.jboss.test.kernel.config.support.PropHolder"
   <property name="list">
      <list elementClass="java.lang.String">
         <value-factory bean="ldap" method="getValue" default="QWERT">
            <parameter>no.such.key</parameter>
         </value-factory>
      </list>
   <property>
</bean>
```

Collections

The `collection` , `list` , `set` and `array` elements are used to enclose collection of values to pass to the bean components as properties or constructor (factory method) parameters. The default types are:

- collection: java.util.ArrayList
- list: java.util.ArrayList
- set: java.util.HashSet
- array: java.lang.Object[]

They all take the same form. So, only `list` is shown here in those examples. You just need to nest `value` elements inside the collection element to specify the contents of the collection. Please note that a `"elementClass"` attribute is required on the collection element, unless you specify explicit types on all the `value` s.

Below is a sample configuration from the `collections` example. It sets a `List` with two elements of mixed types to the `ObjectPrinter.print` property on the `PrintList` named object.

```
<bean name="PrintList"
   class="org.jboss.example.microcontainer.collections.ObjectPrinter">
```

```
   <constructor>
      <parameter>List</parameter>
   </constructor>

   <property name="print">
      <list elementClass="java.lang.String">
         <value>Value of type elementClass</value>
         <value class="java.lang.Integer">4</value>
      </list>
   </property>
</bean>
```

It is also possible to use a `List` as an element inside another `List` . Here is an example.

```
<bean name="Name1" class="com.acme.Example">
   <property name="list">
      <list class="java.util.LinkedList" elementClass="java.lang.String">

         <value>A string</value> <!-- uses elementClass -->

         <value class="java.lang.URL">http://acme.com/index.html</value> <!-- a

         <value><inject bean="SomeBean"/></value> <!-- inject some other bean -

         <value> <!-- a list inside a list -->
            <list elementClass="java.lang.String">
               <value>Another string</value>
            </list>
         </value>

      </list>
   </property>
</bean>
```

Below is what happens inside the microcontainer.

```
Example example = new com.acme.Example();
List list = new LinkedList();
list.add(new String("A string"));
list.add(new URL("http://acme.com/index.html"));
list.add(someBean);
List subList = new ArrayList();
```

```
subList.add(new String("Another string"));
list.add(subList);
element.setList(list);
```

The other type of collection is a map which also covers Properties and Hashtables. The default is `java.util.HashMap` . The `entry` element inside the map differentiates each `key` and `value` pair. For maps there are two default types for the elements: the `keyClass` and `valueClass` . Below is a map sample from the `collections` example.

```
<bean name="PrintMap"
   class="org.jboss.example.microcontainer.collections.ObjectPrinter">

   <constructor>
      <parameter>Map</parameter>
   </constructor>

   <property name="print">
      <map keyClass="java.lang.String" valueClass="java.lang.String">
         <entry>
            <key>Key1 of type keyClass</key>
            <value>Value1 of type valueClass</value>
         </entry>

         <entry>
            <key>Key2 of type keyClass</key>
            <value class="java.lang.Integer">4</value>
         </entry>

         <entry>
            <key class="java.lang.Long">4</key>
            <value>Value of type valueClass</value>
         </entry>
      </map>
   </property>

</bean>
```

When using collections in your bean configuration, this is the order Microcontainer will use for new instantiation or using already existing collection instance.

- if collection's class attribute is defined, a new instance of that class will be instantiated
- if already instantiated collection exists and is available via getter, this instance will be used - unless you define preinstantiate attribute on the property to false
- if the collection references some non interface class, we'll try to instantiate a new instance of that class

- default collection instance will be used - Collection and List will use ArrayList, Set uses HashSet and Map will use HashMap

```
<bean name="SimpleBean" class="org.jboss.test.kernel.config.support.Unmodifiabl
   <property name="list" preinstantiate="false">
      <list elementClass="java.lang.String">
         <value>string1</value>
         <value>string2</value>
      </list>
   </property>
</bean>
```

Lifecycle

Anybody familiar with the JBoss JMX microkernel will know about the lifecycle. The microcontainer extends the lifecycle concept to the managed POJOs. A POJO can have the following lifecycle states.

- Not Installed: The POJO has not been described or has been uninstalled.
- Pre Install: The scoping description has been examined and classloader dependencies determined.
- Described: The POJO's bean description has been examined and dependencies determined.
- Instantiated: All the dependencies have been resolved to construct the bean, these include, the class exists, the constructor parameter injections can be resolved, any factory can be resolved.
- Configured: All the property injections can be resolved, this includes all the dependencies in any collections.
- Create: All the dependent beans have been "created", this includes any injections passed to the create method.
- Start: All the dependent beans have been "started", this includes any injections passed to the start method.
- Installed: The lifecycle is complete.
- *** ERROR ***: Some unexpected error occured, usually due to misconfiguration.

At each stage of the lifecycle, the corresponding method in the bean class is automatically called by the Microcontainer, so that you can programatically control how the objects behave throughout its lifecycle. For instance, the `start()` method in the bean class is called when the bean enters the `Start` state. Below is the `LifecycleBean` class from the `lifecycle` example.

```
public class LifecycleBean
{
```

```
   String name;

   public LifecycleBean(String name)
   {
      this.name = name;
      System.out.println("LifecycleBean() " + this);
   }

   public void create()
   {
      System.out.println("create: " + this);
   }

   public void start()
   {
      System.out.println("start: " + this);
   }

   public void stop()
   {
      System.out.println("stop: " + this);
   }

   public void destroy()
   {
      System.out.println("destroy: " + this);
   }

   public String toString()
   {
      return name;
   }
}
```

The `depends` element allows two beans to perform two phase startup processing like the JMX microkernel.

```
<bean name="Lifecycle1"
   class="org.jboss.example.microcontainer.lifecycle.LifecycleBean">

   <constructor>
      <parameter>Lifecycle1</parameter>
   </constructor>

   <depends>Lifecycle2</depends>

</bean>
```

```
<bean name="Lifecycle2"
   class="org.jboss.example.microcontainer.lifecycle.LifecycleBean">

   <constructor>
      <parameter>Lifecycle2</parameter>
   </constructor>

</bean>
```

The microcontainer resolves the dependency and starts both beans in the appropriate order. Below is the console output when you run the `lifecycle` example. It shows when various lifecycle methods are called when the bean enters those states.

```
run:
[java] LifecycleBean() Lifecycle1
[java] LifecycleBean() Lifecycle2
[java] create: Lifecycle2
[java] create: Lifecycle1
[java] start: Lifecycle2
[java] start: Lifecycle1
[java] stop: Lifecycle1
[java] stop: Lifecycle2
[java] destroy: Lifecycle1
[java] destroy: Lifecycle2
```

The `create()` , `start()` , `stop()` and `destroy()` methods can be overridden with parameters passed to them. Below is an example on how to override the `create()` method via the `jboss-beans.xml` configuration file.

```
public class Example
{
   public void initialize(Object someObject) {}
}

<bean name="Name1" class="com.acme.Example">
   <create method="initialize">
      <parameter><inject bean="SomeBean"/></parameter>
   </create>
</bean>
```

They can also be ignored. See the `jboss-beans.xml` with ignore attribute set to true on the start element.

```
public class Example
{
   public void start() {}
}

<bean name="Name1" class="com.acme.Example">
   <start ignore="true"/>
</bean>
```

In this case Microcontainer would not invoke start method when passing over Start state.

Installation

As of 2.0.0, you can provide generic install/uninstall actions. Allowing you to dynamically setup repositories. Note the use of `this` to pass yourself as a parameter. If you exlude the bean name on the action, the operation is performed on yourself.

```
<bean name="name1" class="java.util.Timer"/>

<bean name="name2" ...>
   <install bean="name1" method="schedule">
      <parameter><this/></parameter>
      <parameter>100</parameter>
      <parameter>10000</parameter>
   </install>

   <uninstall method="cancel"/>

</bean>

// Install
name1 = new Timer();
name2 = ...;
name1.schedule(Name2, 100, 10000);

// Uninstall
name2.cancel();
```

ClassLoader

The Microcontainer supports configuration of the classloader at either the deployment or bean level. The classloader element has three alternatives.

```
<!-- deployment level - applies to all beans in the deployment -->
<deployment>
   <classloader><inject bean="ClassLoaderName"/></classloader>

   <!-- bean level -->
   <bean name="Name2" ...>
      <classloader><inject bean="ClassLoaderName"/></classloader>
   </bean>

   <!-- bean level will use any deployment level classloader -->
   <bean name="Name2" ...>
   </bean>

   <!-- bean level as null to not use any deployment level classloader -->
   <bean name="Name2" ...>
      <classloader&gt;<null/></classloader>
   </bean>

</deployment>
```

Annotations via XML

With the new Microcontainer 2.0 we've added support for annotations via XML. Meaning you can have annotation support even in Java 1.4 and before, depending on the retrowoven Microcontainer project. Each annotation is set per bean instance and you can set annotations on the following types:

- deployment (all beans inherit this annotations)
- bean
- constructor
- lifecycle
- install / uninstall
- callback
- annotated value

Let's see some use cases on the examples below.

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

   <annotation>@org.jboss.test.kernel.deployment.xml.support.Annotation1</annot
```

```
   <annotation>@org.jboss.test.kernel.deployment.xml.support.AnnotationWithAttr

   <bean name="bn1" class="org.jboss.test.Example1">
      <annotation>@org.jboss.test.kernel.deployment.xml.support.Annotation2</an
      <annotation>@org.jboss.test.kernel.deployment.xml.support.Annotation3</an
   </bean>

   <bean name="bn2" class="org.jboss.test.Example2">
      <constructor>
         <annotation>@org.jboss.test.kernel.deployment.xml.support.Annotation3<
      </constructor>
   </bean>

   <bean name="bn3" class="org.jboss.test.Example3">
      <install method="someMethod">
         <annotation>@org.jboss.test.kernel.deployment.xml.support.Annotation1<
      </install>
   </bean>

   <bean name="bn4" class="org.jboss.test.Example4">
      <create>
         <annotation>@org.jboss.test.kernel.deployment.xml.support.Annotation1<
      </create>
   </bean>

   <bean name="bn5" class="org.jboss.test.Example5">
      <property name="PropertyName">
         <annotation>@org.jboss.test.kernel.deployment.xml.support.Annotation1<
         <value>123</value>
      </property>
   </bean>

</deployment>
```

The only issue with having annotations defined via XML is that they are per instance basis - as already mentioned, not added at compile time. In this case you need an external mechanism in order to do the annotation inspection. In Microcontainer this is its metadata repository. See how this is done in Reference Manual, TODO.

Alias

Mostly each bean will have one unique name, but sometimes it is useful to be able to have an extra name. The extra ones are considered aliases.

```
<bean xmlns="urn:jboss:bean-deployer:2.0" class="org.acme.Example">
   <alias>SimpleAlias</alias>
</bean>
```

As with all String values you can have System property replacement as part of actual alias value. Additionally you can specify if this replacement is ignored. Or you can even convert final alias String value into any other class supported by PropertyEditors by setting class attribute on alias element. Lets see this configuration on the examples below.

```
<bean xmlns="urn:jboss:bean-deployer:2.0" class="org.acme.Example">
   <alias>${example.cluster.name}</alias>
</bean>

<bean xmlns="urn:jboss:bean-deployer:2.0" class="org.acme.Example">
   <alias replace="false">X${alias.test.name}X</alias>
</bean>

<bean xmlns="urn:jboss:bean-deployer:2.0" class="org.acme.Example">
   <alias class="java.lang.Integer">12345</alias>
</bean>
```

Having to specify all aliases when the bean is actually defined is not always adequate however. It is sometimes desirable to introduce an alias for a bean which is defined elsewhere. In XML-based configuration metadata this may be accomplished via the use of the standalone `<alias/>` element.

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

   <bean name="Bean1" class="java.lang.Object"/>

   <bean name="Bean2" class="java.lang.Object"/>

</deployment>

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

   <bean name="Xyz1" class="java.lang.Object">
      <property name="locker1"><inject name="Lock1"></property>
      <property name="locker2"><inject name="Lock2"></property>
   </bean>

   <alias name="Bean1">Lock1</alias>

   <alias name="Bean2">Lock2</alias>
</deployment>
```

Newly added alias defined as a element on the deployment acts as a full dependency item. Meaning it won't get registered (installed) until the original bean is installed. What this means that you can have original beans defined in one deployment and aliases in another, and the order of deployment doesn't matter. When the original bean is uninstalled, the alias gets unregistered (uninstalled) as well. When the alias gets unregistered, all beans that depend on the alias get undeployed as well, regardless if the original bean is still present.

Supply and Demand

There is another useful definition of loosely coupled dependency. Each bean can define which beans should be installed for it to move to certain state.

```
<bean name="TM" class="com.acme.SomeSingleton">
   <property name="host">http://www.jboss.org</property>
</bean>

<bean name="SM" class="com.acme.AnotherSingleton">
   <property name="treadSize">10</property>
</bean>

<bean name="Name2" class="com.acme.Example">
   <property name="threadPool"><inject bean="pool"/></property>
   <demand state="Start">TM</demand>
   <demand state="Configure">SM</demand>
</bean>

// Example class
public class Example
{
   public void start()
   {
      SomeSingleton tm = ...; // should be there!
   }

   public void setThreadPool(ThreadPool pool)
   {
      AnotherSingleton config = ...; // should be there
      pool.setThreadSize(config.getThreadSize());
   }

}
```

On the other hand each bean can also provide additional information what it supplies, apart from it self. The actual supply doesn't need to be binded to Microcontainer in any way. More about this feature with the actual example.

```
public class FooBarBinder extends JndiBinder
{
   public void start()
   {
      FooBar foobar = new FooBar();
      bindToJndi("fbName", foobar);
   }

}

public class FooBarConsumer
{
   public FooBarConsumer()
   {
      FooBar foobar = ...; // get it from jndi. should be there
   }
}

<bean name="provider" class="com.acme.FooBarBinder">
   <supply>foobar</supply>
</bean>

<bean name="consumer" class="com.acme.FooBarConsumer">
   <demand state="Instantiate">foobar</demand>
</bean>
```

Contextual awareness

With existing annoyance of using huge amounts of xml to sometimes wire up trivial beans, IoC containers provide a simpler solution called autowiring. Beans, specially singletons, can often be wired with a simple class type knowledge. But with huge systems you should use this feature with care, since it can easly get broken with addition of another bean that would satisfy contextual dependency. Existing 2.0.0 version supports three ways of autowiring:

- full bean autowiring
- property / parameter injection
- callback injection

On the other hand, if you don't want for a particular bean to be involved as a candidate for autowiring, you can set bean element's attribute `autowire-candidate` to false.

```
<bean name="ignored" class="com.acme.Example" autowire-candidate="false" />
```

TODO for bean autowiring.

```
<bean name="autowired" class="com.acme.Example" autowire="true" />
```

When doing a propery / parameter injection of some existing bean, you can omit the bean attibute, meaning that Microcontainer will do the injection based on the information from the property / parameter class type. If no such information is available, exception will be thrown. If no such bean is eventually available, you can define the behavior after validation. For Strict option the deployment will fail, for Callback see the callback injection below. There is also a way of doing contextual injection based on property name. Property name will be matched against newly wired bean's name.

```
<bean name="propInj" class="com.acme.ThreadPoolImpl"/>

<bean name="propInj" class="com.acme.Example">
   <property name="threadPool"><inject/></property>
</bean>

<bean name="paramInj" class="com.acme.Example">
   <constructor>
      <parameter name="threadPool"><inject/></parameter>
   </constructor>
</bean>

public class Example
{
   protected ThreadPool pool;

   public Example() {}

   public Example(ThreadPool pool)
   {
      this.pool = pool;
   }

   public void setThreadPool(ThreadPool pool)
   {
      this.pool = pool;
   }

}
```

Sometimes it is useful to be notified of certain beans being installed / uninstalled by inspecting their class type. In Microcontainer we call this callbacks. You can define which methods should be used for callback resolution on a particular bean. Usually method name is sufficient, in the case of having multiple methods

with the same name, you can narrow down the exact method by specifying class type to use for contextual lookup.

```
<bean name="editorHolder" class="com.acme.Example">
   <incallback method="addEditor" />
   <uncallback method="removeEditor" />
</bean>

<bean name="fbEditor" class="com.acme.FooEditor"/>

public class Example
{
   protected Set<Editor> editors = new HashSet<Editor>();

   public Example() {}

   public void addEditor(Editor editor)
   {
      editors.add(editor);
   }

   public void removeEditor(Editor editor)
   {
      editors.remove(editor);
   }
}
```

We can do the same callback injection on a property. And there is also an interesting feature available, though it will be probabaly rarely used. You can define a Cardinality as a condition when that actual dependency is satisfied and injection can take place. The following examples shows how we need at least 2 Editor instances to satisfy callback dependency.

```
<bean name="editorHolder" class="com.acme.Example">
   <incallback property="editors" cardinality="2..n" />
   <uncallback property="editors" />
</bean>

<bean name="fbEditor" class="com.acme.FooEditor"/>

<bean name="xyzEditor" class="com.acme.XYZEditor"/>

public class Example
{
   protected Set<Editor> editors;
```

```
   public Example() {}

   public void setEditors(Set<Editor> editors)
   {
      this.editors = editors;
   }
}
```

While using callbacks on the Collection subclasses, currently only basic Collection subinterfaces are supported: List, Set and Queue. See BasicCollectionCallbackItemFactory for more details. But you can change the Collection callback behaviour by providing your own CollectionCallbackItemFactory. You do this by setting System property `org.jboss.dependency.collectionCallbackItemFactory` and putting CollectionCallbackItemFactory implementation fully qualified class name as a value.

Injection from context

Even though Microcontainer is aimed at POJOs, it doesn't mean you are not able to easily get into the underlying architecture of its state machine. Each bean is represented in Microcontainer's state machine (Controller) as a ControllerContext. ControllerContext holds various information in order to consistently move between lifecycle states, resolve dependencies, apply instance aspects, annotations, metadata ...

With already known injection concept (`<inject />`), now you can also inject some information from the context. This is what you can currently get a hold on from the underlying context:

- name - `<inject fromContext="name"/>`
- aliases - `<inject fromContext="aliases"/>`
- metadata - `<inject fromContext="metadata"/>`
- beaninfo - `<inject fromContext="beaninfo"/>`
- scope - `<inject fromContext="scope"/>`
- id - `<inject fromContext="id"/>`
- context itself - `<inject fromContext="context"/>`

All this information is wrapped into unmodifiable objects, preventing user to add or set additional information outside Microcontainer's Controller. As we already know, `<inject/>` element can take bean attribute. In this case, if bean attribute is set, we are not targeting bean's underlying context, but context that coresponds to set bean attribute.

```
<bean name="sndBean" class="org.jboss.test.kernel.deployment.support.NameAwareB
   <property name="beaninfo"><inject bean="otherBean" fromContext="beaninfo"/><
</bean>
```

In this example we inject otherBean's beaninfo information into sndBean bean.

Annotations support

We learned how to configure our POJOs with Microcontainer's XML elements. In today's development most of the configuration is done with annotations. Microcontainer is no exception.

Almost all of the features available in XML are also present in Microcontainer's annotations support, except for the classloading configuration, which for obvious reasons doesn't have its equivalent (class needs to be loaded before we can check for annotations, right). Let's now look at the annotations supported and some of its caveats.

Like in most other configurations XML and predetermined bean metadata override annotations. This is current list of supported annotations and their XML equivalents. Later we will show how you can extend support for your own annotations directly into Microcontainer's configuration.

- @Aliases - <alias>

```
/**
 * The aliases.
 * Equivalent to deployment's alias element.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin</
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface Aliases
{
   /**
    * Get aliases array.
    *
    * @return the aliases
    */
   String[] value();

   /**
    * Do system property replace.
    *
    * @return true to replace system property, false otherwise
    */
   boolean replace() default true;
}
```

Example:

```
@Aliases({"MyAlias", "YetAnotherSimpleBean"})
public class AliasSimpleBeanImpl extends SimpleBeanImpl
```

- @ArrayValue - <array>

@CollectionValue - <collection>

@ListValue - <list>

@SetValue - <set>

```
/**
 * Injecting array value.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin</
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface ArrayValue
{
   /**
    * Set the array class.
    *
    * @return array class name
    */
   String clazz() default "";

   /**
    * Array's element class.
    *
    * @return element class name
    */
   String elementClass() default "";

   /**
    * Get the values.
    *
    * @return the values
    */
   Value[] value();
}
```

Example:

```
@ArrayValue(
      value = {
         @Value(string = @StringValue("string1")),
         @Value(string = @StringValue("string2")),
         @Value(string = @StringValue("string2")),
         @Value(string = @StringValue("string1"))
      },
      elementClass = "java.lang.String",
      clazz = "[Ljava.lang.Comparable;"
)
public void setArray(Object[] array)
{
```

```
      super.setArray(array);
   }

   @CollectionValue(
         value = {
            @Value(string = @StringValue("string1")),
            @Value(string = @StringValue("string2")),
            @Value(string = @StringValue("string2")),
            @Value(string = @StringValue("string1"))
         },
         elementClass = "java.lang.String",
         clazz = "java.util.ArrayList"
   )
   public void setCollection(Collection collection)
   {
      super.setCollection(collection);
   }
```

- @Constructor - <constructor>

```
/**
 * Mark the constructor used to instantiate bean.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin</
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.CONSTRUCTOR})
public @interface Constructor
{
}
```

Example:

```
@Constructor
public TestConstructorBean(@Inject(bean = "testBean") TestBean
{
   ...
}
```

- @Create - <create>

  @Start - <start>

  @Stop - <stop>

  @Destroy - <destroy>

```
/**
 * Mark lifecycle create method.
```

```
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Create
{
   /**
    * Is this lifecycle callback ignored.
    *
    * @return ignored
    */
   boolean ignored() default false;
}
```

Example:

```
@Create
public void notCreate(@Inject(bean = "Name1") SimpleBeanWithLi
{
   super.notCreate(bean);
}

@Start
public void notStart(@Inject(bean = "Name2")SimpleBeanWithLifec
{
   super.notStart(bean);
}

@Stop
public void notStop(@Inject(bean = "Name3")SimpleBeanWithLifecy
{
   super.notStop(bean);
}

@Destroy
public void notDestroy(@Inject(bean = "Name4")SimpleBeanWithLi
{
   super.notDestroy(bean);
}
```

- @Demands - <demands>

```
/**
 * The demands.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
```

```
public @interface Demands
{
   /**
    * Get demands.
    *
    * @return the demands
    */
   Demand[] value();
}
```

Example:

```
@Demands({@Demand("WhatIWant")})
public class DemandSimpleBeanImpl extends SimpleBeanImpl
```

- @Depends - <depends>

```
/**
 * The depends values.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface Depends
{
   /**
    * Get the depends values.
    *
    * @return the depends
    */
   String[] value();
}
```

Example:

```
@Depends({"Name1"})
public class DependSimpleBeanWithLifecycle extends SimpleBeanW
```

- @ExternalInstalls - <install bean="..." method="..."/>

  @ExternalUninstalls - <uninstall bean="..." method="..."/>

```
/**
 * Array of external installs.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
```

```
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface ExternalUninstalls
{
   /**
    * Get external installs.
    *
    * @return the external installs
    */
   ExternalInstall[] value();
}
```

Example:

```
@ExternalInstalls({@ExternalInstall(bean = "Name1", method = "a
@ExternalUninstalls({@ExternalInstall(bean = "Name1", method =
public class ExternalInstallSimpleBeanImpl implements Serializa
```

- @Factory - <constructor><factory ...>

```
/**
 * Define constructor factory.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin</A
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface Factory
{
   /**
    * Get the factory.
    *
    * @return the factory value
    */
   Value factory() default @Value();

   /**
    * Get the factory class.
    *
    * @return the factory class
    */
   String factoryClass() default "";

   /**
    * Get the factory method.
    *
    * @return the factory method
    */
   String factoryMethod();
```

```
   /**
    * Get parameters.
    *
    * @return the parameters
    */
   Value[] parameters() default {};
}
```

Example:

```
@Factory(
      factory = @Value(javabean = @JavaBeanValue("org.jboss.tes
      factoryMethod = "createSimpleBean",
      parameters = {@Value(string = @StringValue("Factory Value
)
public class FromFactoryWithParamSimpleBean extends SimpleBean
```

- @FactoryMethod - <constructor><factory factoryMethod="..." ...>

```
/**
 * Mark static method as factory method.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<A
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface FactoryMethod
{
}
```

Example:

```
@FactoryMethod
public static SimpleInject getInstance(@NullValue Object someNu
{
   return new SimpleInject();
}
```

- @Inject - <inject>

```
/**
 * Beans when injected by class type are by default changed to
 * state - if not yet configured.
 * You can change this behavior by setting state.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<A
```

```
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface Inject
{
   /**
    * Get bean.
    * Default is no bean.
    *
    * @return bean name
    */
   String bean() default "";

   /**
    * Get property.
    * Default is no property.
    *
    * @return property name
    */
   String property() default "";

   /**
    * Get when required.
    *
    * @return when required.
    */
   String whenRequired() default "";

   /**
    * Get dependent state.
    * Default is Installed.
    *
    * @return dependent state.
    */
   String dependentState() default "Installed";

   /**
    * Get injection type.
    * Default is by class.
    *
    * @return injection type
    */
   InjectType type() default InjectType.BY_CLASS;

   /**
    * Get injection option.
    * Default is Strict.
    *
    * @return injection option
    */
   InjectOption option() default InjectOption.STRICT;

   /**
    * Get from context injection.
```

```
     *
     * @return from context type
     */
    FromContext fromContext() default FromContext.NONE;

    /**
     * Is this @Inject valid.
     * Used with @Value.
     *
     * @return is this instance valid
     */
    boolean valid() default true;
}
```

Example:

```
@Inject(bean = "Name1")
public void setSimpleBean(SimpleBean bean)
{
   super.setSimpleBean(bean);
}

@Start
public void startMeUp(@Inject(bean = "lifecycleBean") TestBean
{
   test = bean;
   intVF =- value;
}
```

- @Install - <incallback>

  @Uninstall - <uncallback>

```
/**
 * Install callback.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface Install
{
   /**
    * Get the cardinality.
    * Default is no cardinality.
    *
    * @return cardinality
    */
   String cardinality() default "";

   /**
```

```
     * Get when required.
     * Default is Configured.
     *
     * @return when required.
     */
    String whenRequired() default "Configured";

    /**
     * Get dependent state.
     * Default is Installed.
     *
     * @return dependent state.
     */
    String dependentState() default "Installed";
}
```

Example:

```
@Install
public void addDeployer(MyDeployer deployer)
{
   if (deployers == null)
      deployers = new HashSet<MyDeployer>();
   deployers.add(deployer);
}

@Uninstall
public void removeDeployer(MyDeployer deployer)
{
   deployers.remove(deployer);
}
```

- @InstallMethod - <install>

  @UninstallMethod - <uninstall>

```
/**
 * Internal installation method.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface InstallMethod
{
   /**
    * Get dependant state.
    *
    * @return the dependant state
    */
   String dependantState() default "";
```

```
}
```

Example:

```
@InstallMethod
public void install()
{
   installed = true;
}

@UninstallMethod
public void uninstall()
{
   installed = false;
}
```

- @JavaBeanValue - <javabean>

```
/**
 * Java bean value.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface JavaBeanValue
{
   /**
    * Get java bean class name.
    * Must have default constructor.
    *
    * @return the class name
    */
   String value() default "";
}
```

Example:

```
@Factory(
      factory = @Value(javabean = @JavaBeanValue("org.jboss.te
      factoryMethod = "createSimpleBean"
)
```

- @MapValue - <map>

```
/**
 * Map value injection.
```

```
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin</
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface MapValue
{
   /**
    * Get the map class name.
    *
    * @return the map class name
    */
   String clazz() default "";

   /**
    * Get the key class name.
    *
    * @return the key class name
    */
   String keyClass() default "";

   /**
    * Get the value class name.
    *
    * @return the value class name
    */
   String valueClass() default "";

   /**
    * Get the entries.
    *
    * @return the entries
    */
   EntryValue[] value();
}
```

Example:

```
@Constructor
public AnnotatedLDAPFactory(
      @MapValue(
            keyClass = "java.lang.String",
            valueClass = "java.lang.String",
            value = {
               @EntryValue(
                  key = @Value(string = @StringValue("foo.bar.
                  value = @Value(string = @StringValue("QWERT"
               ),
               @EntryValue(
                  key = @Value(string = @StringValue("xyz.key"
                  value = @Value(string = @StringValue(" QWERT
               )
```

```
            }
      )
      Map<String, String> map
)
{
   super(map);
}
```

- @NullValue - <null>

```
/**
 * Null value.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface NullValue
{
   /**
    * Is valid.
    * Used in @Parameter and @value to define
    * unused value.
    *
    * @return is value valid
    */
   boolean valid() default true;
}
```

Example:

```
@UninstallMethod
public void withUninstall(@ThisValue SimpleInject me, @NullValu
{
   System.out.println(me == this);
   System.out.println("plainNull = " + plainNull);
}
```

- @StringValue - <value>somestring</value>

```
/**
 * String value.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface StringValue
{
```

```
   /**
    * Get the value.
    *
    * @return the value
    */
   String value();

   /**
    * Get type.
    *
    * @return the type
    */
   String type() default "";

   /**
    * Do replace with system properties.
    *
    * @return true for replace with system properties, false ot
    */
   boolean replace() default true;
}
```

Example:

```
@Constructor
public ParamIntConstructorAnnBean(@StringValue("7") Integer str
{
   super(string);
}
```

- @Supplys - <supply>

```
/**
 * The supplys.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin</A
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface Supplys
{
   /**
    * Get supply values.
    *
    * @return the supplys
    */
   String[] value();
}
```

Example:

```
@Supplys({"WhatIWant"})
public class SupplyPlainDependecySimpleBeanImpl extends PlainDe
```

- @ThisValue - <this/>

```
/**
 * This value.
 * Get the underlying target.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<A
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface ThisValue
{
   /**
    * Is instance valid.
    *
    * @see @org.jboss.beans.metadata.api.annotations.Value
* @see @org.jboss.beans.metadata.api.annotations.Parameter
* @return true for valid
    */
   boolean valid() default true;
}
```

Example:

```
@InstallMethod
public void whenInstalled(@ThisValue SimpleInject me, @NullValu
{
   System.out.println(me == this);
   System.out.println("plainNull = " + plainNull);
}
```

- @ValueFactory - <value-factory>

```
/**
 * The value factory.
 *
 * @author <a href="mailto:ales.justin@jboss.com">Ales Justin<A
 */
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD, ElementType.PARAMETER})
public @interface ValueFactory
{
   /**
    * Get the bean.
```

```
     *
     * @return the bean
     */
    String bean();

    /**
     * Get the method.
     *
     * @return the method
     */
    String method();

    /**
     * Get single parameter.
     *
     * @return the single parameter
     */
    String parameter() default "";

    /**
     * Get parameters.
     *
     * @return the parameters
     */
    Parameter[] parameters() default {};

    /**
     * Get default value.
     *
     * @return the default value
     */
    String defaultValue() default "";

    /**
     * Get dependant state.
     *
     * @return the dependant state
     */
    String dependantState() default "Installed";

    /**
     * Get when required state.
     *
     * @return the when required state
     */
    String whenRequiredState() default "Configured";
}
```

Example:

```
@ValueFactory(bean = "ldap", method = "getValue", parameter = "
public void setValue(String value)
```

```
{
   super.setValue(value);
}
```

TODO - extending annotations support.

AOP Configuration and Usage

When writting new version of Kernel, there was always a need for simple usage of advanced AOP features in the Kernel itself. With the version 2.0.0 of Microcontainer there is an elegant way of binding your aspect to wired POJOs, using all the advantages of full dependency state machine. Meaning that even aspect behave as installed services, having full lifecycle and dependency support in the Microcontainer. And it's all a matter of configuration if you want to use aspectized Microcontainer or just plain POJO behaviour. To find out more about JBoss AOP, please see the JBoss AOP documentation

Configuration

To leverage the JBoss AOP integration in the Microcontainer you need to make sure that `jboss-aop-mc-int.jar` is available on the classpath. The Microcontainer will decide how whether to use AOP depending on if classes from this jar are present.

To use the AOP integration we must include the `AspectManager` bean. in our configuration. It is included as follows

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="urn:jboss:bean-deployer:2.0">
   <bean name="AspectManager" class="org.jboss.aop.AspectManager">
      <constructor
         factoryClass="org.jboss.aop.AspectManager"
         factoryMethod="instance"/>
   </bean>

   <!-- declare beans and aspects -->
</deployment>
```

Using Aspects

We can apply aspects to any beans we like. This can either be done by using already woven classes (loadtime or compile-time weaving). If the bean class is not already woven, but should have aspects applied to it, a proxy will be generated. If the class is not woven, field level aspects will not get triggered, you will only get constructor and method-level interception.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment 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="FooAdvice"
               class="org.jboss.test.microcontainer.support.TestAspect"
               method="foo"
               pointcut="execution(* *.SimpleBeanImpl->someMethod(..))">
   </aop:aspect>

   <bean
      name="Intercepted"
      class="org.jboss.test.microcontainer.support.SimpleBeanImpl"/>
   <!-- declare beans and aspects -->
</deployment>
```

```
import org.jboss.aop.joinpoint.Invocation;

public class TestAspect
{
   public static boolean fooCalled;
   public static boolean barCalled;
   boolean shouldInvoke;

   public Object foo(Invocation inv) throws Throwable
   {
      System.out.println("--- foo");
      return inv.invokeNext();
   }
}
```

In the above example, whenever we call the method `someMethod()` in the `Intercepted` bean, we will get intercepted by the `FooAdvice` bean's `foo()` method. The aspect methods must have the signature shown. If the `method` attribute is left out for the `aop:aspect` advice, it will look for an advice method called `invoke()`. Please see the JBoss AOP documentation for more information about advice methods, and the pointcut expressions used to pick out methods/constructors/fields that should have aspects applied.

AOP Lifecycle callbacks

We can also aspectize the installs and uninstalls of a bean. The following snippet shows a AOP lifecycle callback handler that gets triggered once `StartBean` enters the `start` state upon deployment, and when it leaves the `start` state upon undeployment.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment 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:lifecycle-start xmlns:aop="urn:jboss:aop-beans:1.0"
```

```
                name="InstallAdvice"
                class="org.jboss.test.microcontainer.support.LifecycleCallback"
                classes="@org.jboss.test.microcontainer.support.Start">
   </aop:lifecycle-start>

   <bean name="StartBean"
      class="org.jboss.test.microcontainer.support.SimpleBeanImpl">
      <annotation>@org.jboss.test.microcontainer.support.Start</annotation>
   </bean>

</deployment>



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

The `install` and `uninstall` methods are required, and must have the signature shown. The names of these methods can be overridden by passing in the `installMethod` and `uninstallMethod` attributes as part of the `aop:lifecycle-start` tag.

We can also intercept the install/uninstall upon entering/leaving other states, by substituting `aop:lifecycle-start` with one of the following.

- `aop:lifecycle-configure` - Triggered when the target beans enter/leave the `configure` lifecycle state upon deployment/undeployment
- `aop:lifecycle-create` - Triggered when the target beans enter/leave the `create` lifecycle state upon deployment/undeployment
- `aop:lifecycle-describe` - Triggered when the target beans enter/leave the `describe` lifecycle state upon deployment/undeployment
- `aop:lifecycle-install` - Triggered when the target beans enter/leave the `install` lifecycle state upon deployment/undeployment
- `aop:lifecycle-instantiate` - Triggered when the target beans enter/leave the `instantiate` lifecycle state upon deployment/undeployment

Aspect dependencies

Aspects and AOP lifecycle callbacks configured via the microcontainer can have dependencies just as normal beans can have. Beans which have these aspects applied inherit the aspect's dependencies.

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment 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:lifecycle-configure xmlns:aop="urn:jboss:aop-beans:1.0"
               name="JMXAdvice"
               class="org.jboss.aop.microcontainer.aspects.jmx.JMXLifecycleCallbac
               classes="@org.jboss.aop.microcontainer.aspects.jmx.JMX">
      <property name="mbeanServer"><inject bean="MBeanServer"/></property>
   </aop:lifecycle-configure>

   <aop:aspect xmlns:aop="urn:jboss:aop-beans:1.0"
               name="TxAdvice"
               class="org.acme.aspects.TxAspect"
               pointcut="execution(* *->@org.acme.Tx(..))">
      <property name="txManager"><inject bean="TxManager"/></property>
   </aop:aspect>

   <bean name="BeanWithAspects"
      class="org.jboss.test.microcontainer.support.SimpleBeanImpl">
      <annotation>@org.jboss.aop.microcontainer.aspects.jmx.JMX</annotation>
      <annotation>@org.acme.Tx</annotation>
   </bean>

   <bean name="PlainBean"
      class="org.jboss.test.microcontainer.support.SimpleBeanImpl">
   </bean>

</deployment>
```

`JmxAdvice` depends on a bean called `MBeanServer` having been deployed, and `TxAdvice` depends on a bean called `TxManager` having been deployed. `BeanWithAspects` has been annotated with both `@JMX` and `@Tx`. Using this configuration, `BeanWithAspects` gets both `JMXAdvice` and `TxAdvice` applied. `BeanWithAspects` inherits the dependencies of all applied aspects and AOP lifecycle callbacks, so it cannot be started until `JMXAdvice` and `TxAdvice` have their dependencies satisfied. `PlainBean` on the other hand, has no aspects applied, and so has no extra dependencies from the AOP layer.

Java Beans

In Container module there is a simple support for JavaBeans [http://java.sun.com/products/javabeans/] configuration. When we have some existing JavaBeans XML, we can easily port that bean configuration into Microcontainer beans configuration. Or if all we really need is simple configuration, no dependency, JavaBeans configuration is enough. Let see this in the examples below.

```
      <javabean xmlns="urn:jboss:javabean:1.0" class="org.jboss.test.javabean.supp
         <property name="AString">StringValue</property>
         <property name="AByte">12</property>
```

```
   <property name="ABoolean">true</property>
   <property name="AShort">123</property>
   <property name="anInt">1234</property>
   <property name="ALong">12345</property>
   <property name="AFloat">3.14</property>
   <property name="ADouble">3.14e12</property>
   <property name="ADate">Jan 01 00:00:00 CET 2001</property>
   <property name="ABigDecimal">12e4</property>
   <property name="ABigInteger">123456</property>
   <property name="abyte">12</property>
   <property name="aboolean">true</property>
   <property name="ashort">123</property>
   <property name="anint">1234</property>
   <property name="along">12345</property>
   <property name="afloat">3.14</property>
   <property name="adouble">3.14e12</property>
   <property name="ANumber" class="java.lang.Long">12345</property>
   <property name="overloadedProperty">StringValue</property>
   <property name="XYZ">XYZ</property>
   <property name="abc">abc</property>
</javabean>
```

Simple configuration via JavaBean setters.

```
<javabean xmlns="urn:jboss:javabean:2.0"
    class="org.jboss.test.javabean.support.SimpleBean">
    <constructor factoryClass="org.jboss.test.javabean.support.SimpleBeanFac
        factoryMethod="newInstance">
        <property name="anObject" class="java.lang.Object">anObjectValue</pr
        <property name="AString">StringValue</property>
        <property name="AByte">12</property>
        <property name="ABoolean">true</property>
        <property name="ACharacter">x</property>
        <property name="AShort">123</property>
        <property name="anInt">1234</property>
        <property name="ALong">12345</property>
        <property name="AFloat">3.14</property>
        <property name="ADouble">3.14e12</property>
        <property name="ADate">Jan 01 00:00:00 CET 2001</property>
        <property name="ABigDecimal">12e4</property>
        <property name="ABigInteger">123456</property>
        <property name="abyte">12</property>
        <property name="aboolean">true</property>
        <property name="achar">y</property>
        <property name="ashort">123</property>
        <property name="anint">1234</property>
        <property name="along">12345</property>
        <property name="afloat">3.14</property>
        <property name="adouble">3.14e12</property>
        <property name="ANumber" class="java.lang.Long">12345</property>
        <property name="overloadedProperty">StringValue</property>
        <property name="XYZ">XYZ</property>
```

```
        <property name="abc">abc</property>
    </constructor>
</javabean>
```

Constructor configuration via factory class static method call.

```
<bean xmlns="urn:jboss:bean-deployer:2.0" class="org.jboss.acme.Example">
   <property name="PropertyName">
      <javabean xmlns="urn:jboss:javabean:1.0" class="org.jboss.test.kernel.
   </property>
</bean>
```

You can use JavaBean with Microcontainer beans in all cases where you could use plain value.

Spring integration

Spring integration allows Spring XML files to be used for IoC without Spring BeanFactory/ ApplicationContext. This is possible as JBoss Microcontainer supports the same set of IoC features as Spring. All you need to do is declare the right namespace: `urn:jboss:spring-beans:2.0`. See the following example for a simple plain Spring XML example.

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

   <bean id="testBean" class="org.jboss.test.spring.support.SimpleBean">
      <constructor-arg index="2">
         <value>SpringBean</value>
      </constructor-arg>
      <constructor-arg index="0">
         <value>1</value>
      </constructor-arg>
      <constructor-arg index="1">
         <value>3.14159</value>
      </constructor-arg>
      <property name="mylist">
         <list value-type="java.lang.String">
            <value>onel</value>
            <value>twol</value>
            <value>threel</value>
         </list>
      </property>
      <property name="myset">
         <set value-type="java.lang.String">
            <value>ones</value>
            <value>twos</value>
            <value>ones</value>
         </set>
      </property>
      <property name="mymap">
         <map key-type="java.lang.String">
            <entry>
```

```
                <key>
                   <value>test_key</value>
                </key>
                <value type="java.lang.String">myvalue</value>
             </entry>
          </map>
       </property>
    </bean>

</beans>
```

But things would be too easy if we just let get away with plain Spring XML. So what you can do is mix and match Microcontainer beans with Spring beans. It doesn't matter which starting element you choose, as long as your beans have the right namespace. Let see this on the next two examples.

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

   <bean name="oldBean" class="org.jboss.test.spring.support.OldBean">
      <property name="testBean">
         <inject/>
      </property>
   </bean>

   <bean xmlns="urn:jboss:spring-beans:2.0" id="testBean" class="org.jboss.t
      <property name="mylist">
         <list value-type="java.lang.String">
            <value>onel</value>
            <value>twol</value>
            <value>threel</value>
         </list>
      </property>
   </bean>

</deployment>
```

In this example we have a Microcontainer beans at the start (deployment element at the start), and we mix them with Spring beans.

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

   <bean id="testBean" class="org.jboss.test.spring.support.SimpleBean">
      <property name="refBean">
         <ref bean="oldBean"/>
      </property>
   </bean>

   <bean xmlns="urn:jboss:bean-deployer:2.0" name="oldBean" class="org.jboss
      <property name="javaBeanString">JavaBean</property>
   </bean>

</beans>
```

Here we start with Spring XML and add Microcontainer beans.

As you can see, all you need to change from your existing XML, be it Spring or Microcontainer, is the namespace on the staring Spring beans element or bean element.

Chapter 7. New subprojects

With the new Microcontainer 2.0.0 a lot of new features were added to the Microcontainer core to make integration with new JBoss5 easier and cleaner. Some of the core aspects were entirely re-written. You can see these changes in the subprojects / modules on the figure below.

Figure 7.1. Module dependency

Here we can see the modules that make the whole Microcontainer architecture. Most of them are part of Microcontainer project as subprojects, where MBeans resides in JBossAS project and VFS is a separate project.

AOP-MC-int and Spring-int are extension parts of the Core (Container, Dependency, Kernel). Using AOP-MC-int features is a matter of configuration, where Spring-int is a simple addon. All other modules are optional, but you must satisfy the dependencies presented to you by directed arrows.

Chapter 8. MBeans extensions

In previous versions (pre 5.x) of JBoss Application Server core services were implemented as MBeans. To make things a bit easier, not forcing the requrement for a direct implementation of the DynamicMBean interface, we introduced XMBeans [http://docs.jboss.org/jbossas/jboss4guide/r1/html/ch2.chapter.html#d0e3460]. But with the change of underlying Kernel, moving away from JMX MBeanServer to plain POJO Controller, we also let the handling of the existing MBeans to the new Controller. Apart from supporting all of the existing XMBean features, we also ported some of the rich features from the new Microcontainer IoC model to the XBeans. We'll show these features in the examples below.

```
<mbean name="jboss.test:type=BasicMBeanName" code="BasicMBeanCode">
   <attribute name="Attribute">
      <inject bean="TestName" property="getSomething" state="Instantiated"/>
   </attribute>
</mbean>
```

Here we transparently add a dependency on a bean named TestName, which must be, in order to be injected, in the Instantiated state. Once the dependency is satisfied, we actually don't inject the bean itself into the attribute, but we inject the return value of getSomething method being executed on TestName bean instance. Property and state attributes are optional, by default the dependant state is Installed.

```
<mbean name="jboss.test:type=BasicMBeanName" code="BasicMBeanCode">
   <alias>BasicBean</alias>
   <alias>${system.basic.name}</alias>
</mbean>
```

Here we can see that a MBean can also have alias names. When adding alias you can also use System property replacement.

Chapter 9. VFS Configuration and Usage

The virtual file system module provides a read-only abstraction for a file system accessed using a root URI/URL.

Overview

The virtual file system module provides a read-only abstraction for a file system accessed using a root URI/URL. The main VFS classes include:

- org.jboss.virtual.VFS - A factory class for creating and finding VirtualFiles.
- org.jboss.virtual.VirtualFile - the encapsulation of an entry in a virtual file system.
- org.jboss.virtual.VirtualFileFilter/VirtualFileFilterWithAttributes - filters that can be used to restrict the VirtualFiles matching a search.
- org.jboss.virtual.VisitorAttributes - attributes that control how a visitation of a VFS root is performed.
- org.jboss.virtual.VirtualFileVisitor - a callback interface used with the VFS/VirtualFile.vist methods.
- org.jboss.virtual.plugins.vfs.helpers.SuffixMatchFilter - a VirtualFileFilter implementation that accepts any file that ends with one of the filter suffixes.

VFS Examples

Example 9.1. Getting a VFS from a root URL

```
URL rootURL = ...;
VFS vfs = VFS.getVFS(rootURL);
VirtualFile jar = vfs.findChild("outer.jar");
```

Example 9.2. Getting a VFS from a root URI

```
URI rootURI = ...;
VFS vfs = VFS.getVFS(rootURI);
VirtualFile jar = vfs.findChild("outer.jar");
```

Example 9.3. Accessing a jar manifest

```
URI rootURI = ...;
VFS vfs = VFS.getVFS(rootURI);
VirtualFile jar = vfs.findChild("outer.jar");
VirtualFile metaInf = jar.findChild("META-INF/MANIFEST.MF");
InputStream mfIS = metaInf.openStream();
Manifest mf = new Manifest(mfIS);
mfIS.close();
Attributes mainAttrs = mf.getMainAttributes();
String version = mainAttrs.getValue(Attributes.Name.SPECIFICATION_V
```

Example 9.4. Finding all .class files under a VFS root

```
URI rootURI = ...;
VFS vfs = VFS.getVFS(rootURI);
SuffixMatchFilter classVisitor = new SuffixMatchFilter(".class", Vi
List<VirtualFile> classes = vfs.getChildren(classVisitor);
int count = 0;
for (VirtualFile cf : classes)
{
...
}
```

org.jboss.virtual Classes

```
         * @throws IllegalStateException if the root is a leaf node
         */
        public 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 root is a leaf node
         */
        public List<VirtualFile> getChildren(VirtualFileFilter filter) throws IOExcepti
        {
        }

        /**
         * 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 root is a leaf node
         */
        public 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 root is a leaf node
         */
        public List<VirtualFile> getChildrenRecursively(VirtualFileFilter filter) throw
        {
        }

        /**
         * Visit the virtual file system from the root
         *
         * @param visitor the visitor
         * @throws IOException for any problem accessing the VFS
         * @throws IllegalArgumentException if the visitor is null
         * @throws IllegalStateException if the root is a leaf node
         */
        public void visit(VirtualFileVisitor visitor) throws IOException
        {
           ...
        }

    }
```

Example 9.6. org.jboss.virtual.VirtualFile

```
   }

   /**
    * 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
    */
   public List<VirtualFile> getChildrenRecursively() throws IOException
   {
      return getChildrenRecursively(null);
   }

   /**
    * 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
    */
   public List<VirtualFile> getChildrenRecursively(VirtualFileFilter filter) throw
   {
...
   }

   /**
    * 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 or it is a leaf node
    */
   public void visit(VirtualFileVisitor visitor) throws IOException
   {
...
   }

   /**
    * Find a child
    *
    * @param path the path
    * @return the child
    * @throws IOException for any problem accessing the VFS (including the child d
    * @throws IllegalArgumentException if the path is null
    * @throws IllegalStateException if the file is closed or it is a leaf node
    */
   public VirtualFile findChild(String path) throws IOException
   {
...
   }

}
```

Example 9.7. org.jboss.virtual.VisitorAttributes

```
   /**
    * Set the includeRoot.
    *
    * @param includeRoot the includeRoot
    * @throws IllegalStateException if you attempt to modify one of the preconfigu
    */
   public void setIncludeRoot(boolean includeRoot)
   {
      this.includeRoot = includeRoot;
   }

   /**
    * Whether to ignore individual errors<p>
    *
    * Default: false
    *
    * @return the ignoreErrors.
    */
   public boolean isIgnoreErrors()
   {
      return ignoreErrors;
   }

   /**
    * Set the ignoreErrors.
    *
    * @param ignoreErrors the ignoreErrors.
    * @throws IllegalStateException if you attempt to modify one of the preconfigu
    */
   public void setIgnoreErrors(boolean ignoreErrors)
   {
      this.ignoreErrors = ignoreErrors;
   }

   /**
    * Whether to include hidden files<p>
    *
    * Default: false
    *
    * @return the includeHidden.
    */
   public boolean isIncludeHidden()
   {
      return includeHidden;
   }

   /**
    * Set the includeHidden.
    *
    * @param includeHidden the includeHidden.
    * @throws IllegalStateException if you attempt to modify one of the preconfigu
    */
   public void setIncludeHidden(boolean includeHidden)
   {
      this.includeHidden = includeHidden;
   }

}
```

Configuration

TODO:jboss-vfs.jar ...

Chapter 10. Deployers module

With re-writing the new application server Kernel from JMX based to POJO oriented, we also re-wrote the whole Deployer architecture which was also based on the old JMX MBeans. Let's first look at the new concepts and then do an example of each new change.

StructureDeployers

Combined with previously mentioned new VFS implementation we introduced a new kind of Deployers, the Structure Deployer. The purpose of Structure Deployer is to recognise the deployment type and prepare this information for the actual Deployers. There are already default implementations for standard types such as JAR, WAR, EAR and specific files. You can simply implement your own StructureDeployer or extend the AbstractStructureDeployer. Or you can use the declarative approach by defining your structure with XML file - META-INF/jboss-structure.xml. This file will be automatically picked up by DeclarativeStructureDeployer. Let's now look at how to use one of the existing StructureDeployer implementations, write our own or use the declarative one.

This is how we can add our own new file support. This one defines that -spring.xml files are also treated as metadata files.

This is a simple implementation for our own deployment structure which holds metadata information in mydata direcotry and has Java classes in myclasses directory.

Here we simply write a plain XML file defining where to look for ... TODO

Deployers

With the old style of JBoss Deployers, a single deployer implementation would handle all the processing for the matching top level deployment unit. This behaviour was completely changed in the new Deployers architecture. Here we have a new way of handling deployment unit, we call it an aspectized deployment, meaning that each deployer implementation does just one thing. This way it is easier to control how much it gets done and even easier to swap out the behaviour. But what is that one thing? We are all familiar with parsing, creating ClassLoaders, installing services, etc. So, basically any part of previous deployment process, we can see as a separate process, parsing the jboss-service.xml file, creating ServiceMetaData, setting up RepositoryCL and finally registring MBeans into the MBeanServer instance.

Let's see this aspectization on the listing below. This is a real example from the current Microcontainer beans deployment.

- Parsing Deployers
  - Turns XML into metadata model
  - e.g. my-beans.xml -> KernelDeployment
- ClassLoading Deployers
  - Creates classloaders from metadata
  - e.g. Uses the information from StructureDeployers

- Component Deployers
  - Splits complicated deployments into units
  - e.g. KernelDeployment -> BeanMetaDatas
- Real Deployers
  - Does the real work of deployment
  - e.g. BeanMetaData -> controller.install()

Attachments

Before we start coding the new Deployers there is another concept that we still need to have a look at. What is the way to store the information between different deployers? We keep this information in so called Attachments. There are two types of attachments, predetermined and transient. e.g. predetermined can be set by ProfileService, where we would get the transient one's from parsing the XML file. You must be aware that a predetermined overriddes transient. This is a simple API to get a hold of the Attachments reference from the underlying DeploymentUnit instance.

```
public interface DeploymentUnit extends MutableAttachments
{
/**
* Get all the metadata for the expected type
*
* @param <T> the type to get
* @param type the type
* @return a set of metadata matching the type
* @throws IllegalArgumentException if the type is null
*/
<T> Set<? extends T> getAllMetaData(Class<T> type);

/**
* Get the transient managed objects
*
* @return the managed objects
*/
MutableAttachments getTransientManagedObjects();

...
}

public interface MutableAttachments extends Attachments
{
/**
* Add attachment
*
* @param name the name of the attachment
* @param attachment the attachment
* @return any previous attachment
* @throws IllegalArgumentException for a null name or attachment
* @throws UnsupportedOperationException when not supported by the impleme
```

```
*/
Object addAttachment(String name, Object attachment);

/**
* Add attachment
*
* @param <T> the expected type
* @param name the name of the attachment
* @param attachment the attachment
* @param expectedType the expected type
* @return any previous attachment
* @throws IllegalArgumentException for a null name, attachment or expecte
* @throws UnsupportedOperationException when not supported by the impleme
*/
<T> T addAttachment(String name, T attachment, Class<T> expectedType);

/**
* Add attachment
*
* @param <T> the expected type
* @param attachment the attachment
* @param type the type
* @return any previous attachment
* @throws IllegalArgumentException for a null name, attachment or type
* @throws UnsupportedOperationException when not supported by the impleme
*/
<T> T addAttachment(Class<T> type, T attachment);

/**
* Remove attachment
*
* @param name the name of the attachment
* @return the attachment or null if not present
* @throws IllegalArgumentException for a null name
* @throws UnsupportedOperationException when not supported by the impleme
*/
Object removeAttachment(String name);

/**
* Remove attachment
*
* @param <T> the expected type
* @param name the name of the attachment
* @return the attachment or null if not present
* @param expectedType the expected type
* @throws IllegalArgumentException for a null name or expectedType
* @throws UnsupportedOperationException when not supported by the impleme
*/
<T> T removeAttachment(String name, Class<T> expectedType);

/**
* Remove attachment
*
* @param <T> the expected type
```

```
* @return the attachment or null if not present
* @param type the type
* @throws IllegalArgumentException for a null name or type
*/
<T> T removeAttachment(Class<T> type);

/**
* Set the attachments
*
* @param map the new attachments a map of names to attachments
* @throws IllegalArgumentException for a null map
*/
void setAttachments(Map<String, Object> map);

/**
* Clear the attachments
*
* @throws UnsupportedOperationException when not supported by the impleme
*/
void clear();

/**
* Get the number of changes that have happened.
*
* @return number of adds/removes that have happened since creation or cle
*/
int getChangeCount();

/**
* Reset the change count to zero.
*/
void clearChangeCount();
}

public interface Attachments extends Serializable
{
/**
* Get all the attachments
*
* @return the unmodifiable attachments
*/
Map<String, Object> getAttachments();

/**
* Get attachment
*
* @param name the name of the attachment
* @return the attachment or null if not present
* @throws IllegalArgumentException for a null name
*/
Object getAttachment(String name);

/**
* Get attachment
```

```
*
* @param <T> the expected type
* @param name the name of the attachment
* @param expectedType the expected type
* @return the attachment or null if not present
* @throws IllegalArgumentException for a null name or expectedType
*/
<T> T getAttachment(String name, Class<T> expectedType);

/**
* Get attachment
*
* @param <T> the expected type
* @param type the type
* @return the attachment or null if not present
* @throws IllegalArgumentException for a null name or type
*/
<T> T getAttachment(Class<T> type);

/**
* Is the attachment present
*
* @param name the name of the attachment
* @return true when the attachment is present
* @throws IllegalArgumentException for a null name
*/
boolean isAttachmentPresent(String name);

/**
* Is the attachment present
*
* @param name the name of the attachment
* @param expectedType the expected type
* @return true when the attachment is present
* @throws IllegalArgumentException for a null name or expectedType
*/
boolean isAttachmentPresent(String name, Class<?> expectedType);

/**
* Is the attachment present
*
* @param type the type
* @return true when the attachment is present
* @throws IllegalArgumentException for a null name or type
*/
boolean isAttachmentPresent(Class<?> type);

/**
* Are there any attachments
*
* @return true if there are any attachments, false otherwise.
*/
boolean hasAttachments();
}
```

Ordering

Since we are now familiar with attachments we can talk about the order of our Deployers. What would be the natural order of our Deployers? Probably the first thing it comes to our mind is a plain number ordering. Old and proven way to do the ordering. But since we waited to introduce the attachments before we talked about ordering there must be something better. We can define the order by simply providing the information about Attachment requirements / demands (inputs) and supplies (outputs). Let's explain this in more detail on the actual example.

```
public MyDeployer()
{
setInputs(SomeMetaData1.class, SomeMetaData2.class);
setOutputs(MyOutput.class);
}
```

Here we can see that our MyDeployer depends on SomeMetaData1 and SomeMetaData2 instance attachments, and provides further MyOutput instance attachment for next deployers to use. So any deployer that provides SomeMetaData1 and SomeMetaData2 instance attachments should be before our MyDeployer and any one that uses MyOutput instance attachment should be after.

Chapter 11. Managed module

Overview

The deployers support a management API based on the org.jboss.managed.api classes, and org.jboss.metatype.api.types, org.jboss.metatype.api.values MetaType/MetaValue classes.

org.jboss.managed.api Classes

The management API provides an abstraction for editing the metadata of a Deployment. The main management classes include:

- org.jboss.managed.api.ManagedObject - is the root interface for a manageable element. Its consists of:
  - a name/name type for a registry/references
  - an attachment name to associate the ManagedObject with a deployment attachment
  - annotations from the metadata making up the ManagedObject
  - the attachment instance
  - the ManagedPropertys for the interface
  - the ManagedOperations for the interface
- org.jboss.managed.api.ManagedProperty is an interface describing a manageable field in a ManagedObject. Its analogous to the JavaBean property/JMX mbean attribute. Its a type safe wrapper around the Fields interface.
- org.jboss.managed.api.ManagedOperation an interface for representing an operation in a management interface.
- org.jboss.managed.api.ManagedParameter – a parameter in a ManagedOperation
- org.jboss.managed.api.Fields is an interface for a collection of named values that are associated with a managed property or operation.
- org.jboss.managed.api.ManagedDeployment is an interface describing a collection of ManagedComponent and structural information about a deployment.
- org.jboss.managed.api.ManagedComponent is an interface that extends ManagedObject to define a runtime component associated with a deployment.
- org.jboss.managed.api.ComponentType – type/subtype qualification of a ManagedComponent
- org.jboss.managed.api.DeploymentTemplateInfo – named collection of ManagedPropertys needed to create a deployment or component

org.jboss.metatype.api.types Classes

The types classes define an abstraction for the types of values found in the ManagedProperty and ManagedOperation interfaces. The types are essentially simplified types that only rely on basic JDK types. The main types classes include:

- org.jboss.metatype.api.types.MetaType<T extends Serializable> - root interface for meta types
- org.jboss.metatype.api.types.CompositeMetaType - a key/value collection type
- org.jboss.metatype.api.types.TableMetaType - a table structure with the rows being a CompositeMetaType.
- org.jboss.metatype.api.types.AbstractMetaType<T extends Serializable> - base abstract class implementing MetaType
- org.jboss.metatype.api.types.AbstractCompositeMetaType - base abstract class implementing CompositeMetaType
- org.jboss.metatype.api.types.ArrayMetaType - an array or Collection of MetaType
- org.jboss.metatype.api.types.EnumMetaType - java enum type or a fixed set of String values.
- org.jboss.metatype.api.types.GenericMetaType - an opaque pass through wrapper for types that don't fit into another MetaType.
- org.jboss.metatype.api.types.SimpleMetaType - a wrapper type for primatives; BigDecimal, BigInteger, Boolean/boolean, Byte/byte, Character/char, Date, Double/double, Float/float, Integer/int, Long/long, Short/short, String, Void/void.

org.jboss.metatype.api.values Classes

The values API provides an value wrappers for the corresponding MetaTypes. The main values classes include:

- org.jboss.metatype.api.values.MetaValue - the base value interface defining a type accessor and clone method.
- org.jboss.metatype.api.values.ArrayValue<T extends Serializable> - a representation of an array or collection of MetaValues. It supports an index getter as well as Iterable<T>.
- org.jboss.metatype.api.values.CompositeValue - A represention of a map of MetaValues keyed by a String. CompositeValueSupport is a concrete implementation of CompositeValue.
- org.jboss.metatype.api.values.EnumValue - A represention of a java.lang.Enum set of values, or a set of Strings. EnumValueSupport is a concrete implementation of EnumValue that represents a single java.lang.Enum value or java.lang.String value.
- org.jboss.metatype.api.values.GenericValue - A represention of a GenericMetaType. GenericValueSupport is a concrete implementation of GenericValue that represents a single java.io.Serializable value.
- org.jboss.metatype.api.values.SimpleValue - the SimpleMetaType value. The org.jboss.metatype.api.values.SimpleValueSupport<T extends Serializable> class provides wrap method for generating the correct SimpleValue from the java value object.
- org.jboss.metatype.api.values.TableValue - the TableMetaType value. The org.jboss.metatype.api.values.TableValueSupport class provides a concrete implementation of TableValue.
- org.jboss.metatype.api.values.MetaValueFactory - a factory for converting a java value into the corresponding MetaValue. It support an org.jboss.metatype.spi.values.MetaValueBuilder plugin for controlling how a given java.lang.Class type is converted into a MetaValue.

- org.jboss.metatype.spi.values.MetaValueBuilder - a plugin for converting a MetaType and object instance into a MetaValue.

org.jboss.managed.api.annotation Annotations

The annotations available for defining the management interfaces include:

- org.jboss.managed.api.annotation.ManagementObject - a class level annotation that identifies a metadata class as a ManagedObject.
- org.jboss.managed.api.annotation.ManagementObjectID - identifies a ManagedObject key/type qualifier source.
- org.jboss.managed.api.annotation.ManagementObjectRef - indicates a property that references another ManagedObject.
- org.jboss.managed.api.annotation.ManagementProperty - annotation for describing a ManagedProperty.
- org.jboss.managed.api.annotation.ManagementComponent - identifies a property as metadata identifying a ManagedComponent.
- org.jboss.managed.api.annotation.ManagementOperation - An annotation for describing a ManagedOperation.
- org.jboss.managed.api.annotation.ManagementParameter - Annotation for documenting a ManagementOperation parameter.

Annotation Examples

Adding ManagedObject Support to Deployers

Deployer implementors create ManagedObjects for their associated metadata by having the Deployer implement the

Chapter 12. Classloader module

Introducing fine grained classloading in Micorcontainer's IoC, better aspect integration, new Chapter 9, *VFS Configuration and Usage* and Chapter 10, *Deployers module* projects, it was all set to also redo the whole classloding layer for the new Kernel and transitively JBoss5 application server. Whole new Classloder module was created to somehow fix the old API mistakes and to allow clean and plugable way for existing extensions of default classloading. This extensions range from old concepts of Servlet classloading, runtime AOP weaving, and all the way to new Chapter 13, *OSGi module*.

When designing new API for Classloading, we wanted to expose as little as possible. The concept there was to hide all the "nitty gritty" details and reduce the amount of API users need to apply in order to get required behaviour. When you were normally dealing with concrete ClassLoader classes, we reduced the implementation detail to only needing to implement your own ClassLoaderPolicy. This includes things like what packages you export, what you import, how to get resources and other things related to defining classes. We'll show the example of the policy later on.

Additionally, the idea of a LoaderRepository is replaced with a ClassLoaderDomain and a ClassLoaderSystem acting as a factory and a repository of domains. Each ClassLoaderSystem has a "default domain".

```
ClassLoaderSystem system = ClassLoaderSystem.getInstance();

// Define classloader policy
MyClassLoaderPolicy myPolicy = ...

// Register with the default domain,
// there are other methods to create
// and register with different domains
ClassLoader cl = system.registerClassLoaderPolicy(myPolicy);
```

This is a simple example of what it falls down to, to create your own Classloader instance that will behave accordingly to the classloader policy.

Chapter 13. OSGi module

We can see an increased demand for OSGi [http://www.osgi.org] technology these days. Developers and users specially like OSGi kind of Classloader wiring. Being able to have Class dependency defined as a versioned package dependency is something we've been lacking for while now. There are new JSRs in the making, that will eventually solve this problem, but OSGi is here and now.

Looking at the OSGi framework we saw a lot similarity with what we already have with Microcontainer. Another aspect of OSGi that is cruical part of the core Framework is Service Registry. A simple lookup pattern that takes Classloading wiring into the consideration when doing Service lookup. While we already have contextual awareness in Microcontainer implementing this was a matter of simple `OSGi Facade` over the existing Microcontainer registry. Together with new Chapter 12, *Classloader module* there is a fully OSGi r4.1 compatible solution available with the 2.0.0 Microcontainer.

OSGi usage in Microcontainer falls down to three different aspects:

- new OSGi Classloader
- declarative or programmatic OSGi services deployment
- OBR (OSGi Bundle Repository) usage

The new OSGi Classloader is used as a default Classloader, it is backward compatible with the existing UCL.

For the existing OSGi declarative services or programmatic usage via BundleActivator, there is a full support of the OSGi core API. But for new service declaration, we encourage people to use much richer Microcontainer IoC features. The Classloading behavior and Service Registry will take new Microcontainer services into consideration.

OBR usage ...

Chapter 14. Reliance modules

In Microcontainer it's all about dependecies. One of the features we promissed it ability to write your own dependency. As a dependency we mean some condition that must be satisfied in order that a state machine let's a node pass to next state. In our case nodes are our beans, which we eventually want to install into Microcontainer's registry.

Almost all component models impose some sort of authentification or authorization. We've added a simple identity module that relies on plain java.security concepts. On top of this we added an extension module where you can define your security requirements via Drools declarative rules. Drools integration is not limited to just security handling, it can be extended to suite any Drools defined rules. More about the subject in the chapters below.

We've also added a jBPM integration to support long lasting state flow definitions. This integration provides a plugable way of notifying state machine nodes that they can move forward to the next state. We'll show a simple human interaction of accepting state change request to move bean into next state.

Reliance identity

Identity ...

Deployment

The `deployment` element acts as a container for many beans that are deployed together.

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

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

<!-- bean part of the deployment -->
<bean .../>

<!-- bean part of the deployment -->
<bean .../>

</deployment&gt;
```

Reliance rules

Drools...

Deployment

The `deployment` element acts as a container for many beans that are deployed together.

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

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

<!-- bean part of the deployment -->
<bean .../>

<!-- bean part of the deployment -->
<bean .../>

</deployment&gt;
```

Reliance jBPM

jBPM ...

Deployment

The `deployment` element acts as a container for many beans that are deployed together.

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

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

<!-- bean part of the deployment -->
<bean .../>

<!-- bean part of the deployment -->
<bean .../>

</deployment&gt;
```

Chapter 15. Guice integration

Guice (pronounced 'juice') is a lightweight dependency injection framework for Java 5 and above, brought to you by Google.

We've written a piece of integration code that bridges the two IoC frameworks together. You can inject Guice defined beans into Microcontainer and/or the other way around. See examples for more details.

In this example we will define Guice module, binding simple Singleton class to one of it's instances. We will then do a contextual lookup for the Singleton instance in SingletonHolder class.

```
AbstractBeanMetaData guicePlugin = new AbstractBeanMetaData("GuicePlugin", G
AbstractConstructorMetaData constructor = new AbstractConstructorMetaData();
AbstractArrayMetaData arrayMetaData = new AbstractArrayMetaData();
final Singleton singleton = new Singleton();
Module module = new AbstractModule()
{
   protected void configure()
   {
      bind(Singleton.class).toInstance(singleton);
   }
};
arrayMetaData.add(new AbstractValueMetaData(module));
constructor.setParameters(Collections.singletonList((ParameterMetaData)new A
guicePlugin.setConstructor(constructor);

public class SingletonHolder
{
   private Singleton singleton;

   @Constructor
   public SingletonHolder(@Inject Singleton singleton)
   {
      this.singleton = singleton;
   }

   public Singleton getSingleton()
   {
      return singleton;
   }
}

ControllerContext holderContext = controller.getInstalledContext("holder");
assertNotNull(holderContext);
SingletonHolder holder = (SingletonHolder)holderContext.getTarget();
assertNotNull(holder);
assertEquals(singleton, holder.getSingleton());
```

The detail that is hidden is in GuiceKernelRegistryEntryPlugin, which acts as a intermediate between Microcontainer's registry and Guice Injector. But all you need to do is register GuiceKernelRegistryEntryPlugin as a POJO into Microcontainer, providing Guice Modules with its constructor.

We can also go the other way around. Injecting named beans into Guice Injector. There are a couple of ways to achieve that. Lets look at them.

```
Injector injector = Guice.createInjector(new AbstractModule()
{
   protected void configure()
   {
      bind(Controller.class).toInstance(controller);
      bind(Singleton.class).toProvider(GuiceIntegration.fromMicrocontaine
      bind(Prototype.class).toProvider(GuiceIntegration.fromMicrocontaine
   }
});
```

```
AbstractBeanMetaData injectorBean = new AbstractBeanMetaData("injector",
AbstractConstructorMetaData constructor = new AbstractConstructorMetaData
constructor.setFactoryClass(GuiceInjectorFactory.class.getName());
constructor.setFactoryMethod("createInjector");
List<ParameterMetaData> parameters = new ArrayList<ParameterMetaData>();
parameters.add(new AbstractParameterMetaData(new AbstractDependencyValueM
AbstractArrayMetaData array = new AbstractArrayMetaData();
array.add(new AbstractValueMetaData(GuiceObject.ALL));
parameters.add(new AbstractParameterMetaData(array));
constructor.setParameters(parameters);
injectorBean.setConstructor(constructor);
controller.install(injectorBean);

ControllerContext injectorContext = controller.getInstalledContext("injec
assertNotNull(injectorContext);
Injector injector = (Injector)injectorContext.getTarget();
```

```
<bean name="injector" class="org.jboss.guice.plugins.GuiceInjectorFactory">
   <constructor factoryClass="org.jboss.guice.plugins.GuiceInjectorFactory"
      <parameter>jboss.kernel:service=Kernel</parameter>
      <parameter>
         <array>
            <bean name="BindAll" class="org.jboss.guice.plugins.AllGuiceObje
               <constructor factoryClass="org.jboss.guice.plugins.AllGuiceOb
            </bean>
         </array>
      </parameter>
   </constructor>
</bean>
```

Here we see three way of usgin Microcontainer beans to do wiring in Guice. The first and second examples are purely programmatic and you need to provide a Controller instance. The third one is how you would bind all existing installed beans into Guice Injector via -beans.xml. Or you can provide a ControllerContextBindFilter instance to the binding methods to filter those beans you want to bind. See API docs for more details.

Chapter 16. Standalone

In Chapter 4, *Starting Examples* , we briefly discussed how to run a Microcontainer application by loading the `StandaloneBootstrap` class, which in turn wraps the `BasicBootstrap` and `BeanXMLDeployer` utility classes. In this chapter, we will look into the source code of `StandaloneBootstrap` and see exactly how it works. While the `StandaloneBootstrap` class is sufficient for most use case scenarios, you do not have to use it. You can trivially write your own class that uses the `BasicBootstrap` and `BeanXMLDeployer` .

```
package org.jboss.kernel.plugins.bootstrap.standalone;

import java.net.URL;
import java.util.Enumeration;
import java.util.List;
import java.util.ListIterator;

import org.jboss.kernel.plugins.bootstrap.basic.BasicBootstrap;
import org.jboss.kernel.plugins.deployment.xml.BeanXMLDeployer;
import org.jboss.kernel.spi.deployment.KernelDeployment;
import org.jboss.util.collection.CollectionsFactory;

public class StandaloneBootstrap extends BasicBootstrap
{
/** The deployer */
protected BeanXMLDeployer deployer;

/** The deployments */
protected List deployments = CollectionsFactory.createCopyOnWriteList();

/** The arguments */
protected String[] args;

/**
* Bootstrap the kernel from the command line
*
* @param args the command line arguments
* @throws Exception for any error
*/
public static void main(String[] args) throws Exception
{
StandaloneBootstrap bootstrap = new StandaloneBootstrap(args);
bootstrap.run();
}

/**
* Create a new bootstrap
*/
public StandaloneBootstrap(String[] args) throws Exception
{
super();
this.args = args;
}
```

```
public void bootstrap() throws Throwable
{
super.bootstrap();

deployer = new BeanXMLDeployer(getKernel());

Runtime.getRuntime().addShutdownHook(new Shutdown());

ClassLoader cl = Thread.currentThread().getContextClassLoader();
for (Enumeration e =
cl.getResources(StandaloneKernelConstants.DEPLOYMENT_XML_NAME);
e.hasMoreElements(); )
{
URL url = (URL) e.nextElement();
deploy(url);
}
for (Enumeration e = cl.getResources("META-INF/" +
StandaloneKernelConstants.DEPLOYMENT_XML_NAME);
e.hasMoreElements(); )
{
URL url = (URL) e.nextElement();
deploy(url);
}

// Validate that everything is ok
deployer.validate();
}

/**
* Deploy a url
*/
protected void deploy(URL url) throws Throwable
{
log.debug("Deploying " + url);
KernelDeployment deployment = deployer.deploy(url);
deployments.add(deployment);
log.debug("Deployed " + url);
}

/**
* Undeploy a deployment
*/
protected void undeploy(KernelDeployment deployment)
{
log.debug("Undeploying " + deployment.getName());
deployments.remove(deployment);
try
{
deployer.undeploy(deployment);
log.debug("Undeployed " + deployment.getName());
}
catch (Throwable t)
{
```

```
log.warn("Error during undeployment: " + deployment.getName(), t);
}
}

protected class Shutdown extends Thread
{
public void run()
{
log.info("Shutting down");
ListIterator iterator =
deployments.listIterator(deployments.size());
while (iterator.hasPrevious())
{
KernelDeployment deployment =
(KernelDeployment) iterator.previous();
undeploy(deployment);
}
}
}
}
```

One way to use this class in your own applications would be:

```
import org.jboss.kernel.plugins.bootstrap.standalone.StandaloneBootstrap

public MyMainClass
{
public static void main(String[] args) throws Exception
{
StandaloneBootstrap.main(args);
// Your stuff here...
}
}
```

So what does the standalone bootstrap do?

First it does the plain bootstrap to get the "kernel" ready. You can think of this a sophisticated form of `ServerLocator` implementation. It then creates a `BeanXMLDeployer` for deploying XML files. Next it adds a shutdown hook, such that deployments are correctly "undeployed" in reverse order to their deployment. Finally, it scans the classpath for `META-INF/jboss-beans.xml` and deploys every instance of that file it finds to populate the "kernel".

You can of course choose not to use this helper class and instead implement your own processing rules.

Chapter 17. Conclusion

The Microcontainer is a powerful replacement for the JBoss's JMX microkernel. It brings the loosely coupled configuration environment of JBoss to POJO environments, adding more control and extra features. Both inside and outside the JBoss Application Server.

Going forward these additional features will allow even more advances, including aspectized deployment, on demand services, versioned deployments, etc.