

# Getting Started with the JBoss Microcontainer

## A Guide for POJO Developers

1.0.0 final

---

# Table of Contents

Target Audience .....	iii
Preface .....	iv
1. Introduction to the JBoss Microcontainer .....	1
2. Installation .....	2
3. Packaging .....	3
4. Examples .....	4
5. Basic Configuration .....	5
5.1. Deployment .....	5
5.2. Bean .....	5
5.3. Construction .....	5
5.4. Factories .....	6
5.5. Properties .....	7
5.6. String Values .....	7
5.7. Injections .....	9
5.8. Collections .....	10
5.9. Lifecycle .....	11
5.10. ClassLoader .....	12
6. Standalone .....	14
7. Conclusion .....	17

---

# Target Audience

This guide is aimed at developers who want to use the Microcontainer in their own applications or use it conjunction with the JBoss Application Server.

It aims at describing the basic elements and steps to configure a POJO deployment.

---

# 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.

## 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 targetted at POJO environments. As such it can be used standalone outside the JBoss Application Server.

This initial release provides a set of features to support POJO deployment in JBoss-4.0.3+ and the bootstrap of services used by the standalone EJB3 distribution.

This document takes you through some example deployments into JBoss-4.0.3+ explaining how to configure POJOs and link them together through injection. Later it shows how to do the same thing outside the application server.

---

# 2

## Installation

First you need download JBoss Microcontainer release from <http://www.jboss.com/downloads/index>

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 JBossAS, you will need to download JBoss-4.0.3 or later <http://www.jboss.com/products/jbossas/downloads>

---

# 3

## Packaging

This chapter describes the packaging. As we will see later in the standalone chapter, this is more of a convention rather than a requirement.

The convention is recommended since it allows "deployments" to be used both standalone and inside JBossAS.

The basic structure of microcontainer deployment is a .beans file (which can also be unpacked - a directory structure that looks the jar file). It also contains a META-INF/jboss-beans.xml to describe what you want it to do. The contents of this xml file are described in the basic configuration chapter.

Finally, it contains the classes and any resources just like any other jar file.

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

If you want to include a .beans 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.beans</service></module>  
</jboss-app>
```

---

# 4

## Examples

The examples folder shows some of the simpler uses of the MicroContainer. Each example is in a subfolder with the following structure:

```
readme.txt - a short description of the example including the expected output
build.xml - the ant build script
src/resources/META-INF/jboss-beans.xml - the MicroContainer configuration for the example
src/main - the java source for the example
```

To run each example, simply:

```
> cd examples/<directory>
> ant
```

To run against the JBoss application server:

```
> <edit> build.properties {change to point at your jboss instance}
> cd examples/<directory>
> ant deploy
followed by
> ant undeploy
to remove the deployment
```

- simple: a simple example to make sure you have everything working
- constructor: simple constructor configuration
- factory: construction using factories
- properties: property configuration
- injection: referencing other beans
- collections: creating collections
- lifecycle: the create/start/stop/destroy lifecycle
- locator: implementing the locator pattern with the Microcontainer

The JBoss Microcontainer uses any logging mechanism supported by `org.jboss.logging`. e.g. if you want to use `log4j`, add `log4j.jar` and a directory containing either a `log4j.properties` or `log4j.xml` to the classpath. The default distribution uses the "null" logging implementation, which is why the examples all use `System.out.println()`.

This just skims the surface of the MicroContainer, showing the most common usecases. Other more complicated examples can be found in the tests (available from cvs).



## Basic Configuration

The microcontainer's main purpose is to allow external configuration of POJOs. In this chapter we will look at the some of the common configurations that can be performed.

### 5.1. Deployment

The deployment 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_1_0.xsd"
             xmlns="urn:jboss:bean-deployer">

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

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

</deployment>
```

### 5.2. Bean

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

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

The example above uses the default constructor of `com.acme.Example` to create a new POJO.

```
new com.acme.Example();
```

It is given the name "Name1" such that it can be referenced elsewhere.

### 5.3. 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.

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

```

}

<bean name="Name1" class=com.acme.Example">
    <constructor>
        <parameter>example string</parameter>
        <parameter>4</parameter>
    </constructor>
</bean>

new com.acme.Example(new String("example string"), 4);

```

Sometimes this does not always work, because there are two constructors with the same number of parameters. In this case, you must specify the types to resolve the ambiguity.

```

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

<bean name="Name1" class=com.acme.Example">
    <constructor>
        <parameter>example string</parameter>
        <parameter class="long">4</parameter>
    </constructor>
</bean>

new com.acme.Example(new String("example string"), (long) 4);

```

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

## 5.4. Factories

Not all classes have public 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 three different types of factory.

Static factory with static method:

```

public class ExampleFactory{
    public static Example createExample() {};
}
public class Example{
}

<bean name="Name1" class=com.acme.Example">
    <constructor factoryClass="com.acme.ExampleFactory" factoryMethod="createExample"/>
</bean>

Example Name1 = ExampleFactory.createExample();

```

Factory instance with static method:

```

public class ExampleFactory{
    public static Example createExample() {};
}
public class Example{
}

```

```
<bean name="Factory1" class=com.acme.ExampleFactory"/>

<bean name="Name1" class=com.acme.Example">
  <constructor factoryMethod="createExample">
    <factory bean="Factory1"/>
  </constructor>
</bean>

ExampleFactory Factory1 = new com.acme.ExampleFactory();
Example Name1 = Factory1.createExample();
```

Factory instance with instance method (also shows parameters):

```
public class ExampleFactory{
    public Example createExample(String string) {};
}
public class Example{
}

<bean name="Factory1" class=com.acme.ExampleFactory"/>

<bean name="Name1" class=com.acme.Example">
  <constructor factoryMethod="createExample">
    <factory bean="Factory1"/>
    <parameter>example string</parameter>
  </constructor>
</bean>

ExampleFactory Factory1 = new com.acme.ExampleFactory();
Example Name1 = Factory1.createExample(new String("example string"));
```

## 5.5. Properties

It is possible to create all objects using factories and constructors, however many people use the `javabeans` or `mbeans` convention where an object is constructed using a default constructor and then configured using properties or attributes.

```
public class Example{
    public String getTitle() {}
    public void setTitle(String string) {}
}

<bean name="Name1" class=com.acme.Example">
  <property name="title">example string</property>
</bean>

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

## 5.6. String Values

So far we have looked at two configurations that require values to be configured. The microcontainer has a number of mechanisms to construct static values.

Null: The null value is trivial, `<null/>`. But, it needs to be differentiated from the string "null".

```
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);
```

PropertyEditor: A java bean 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.

```
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());
```

Override the type: Often the property takes an interface or abstract class, but you want to pass a specific implementation or a subclass.

```
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));
```

## 5.7. Injections

Objects by themselves 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.

```
public class Example{
    public Number getNumber() {}
    public void setNumber(Number number) {}
}

<bean name="Four" class="java.lang.Long">
    <constructor><parameter>4</parameter></constructor>
</bean>

<bean name="Name1" class=com.acme.Example">
    <property name="number"><inject bean="Four"/></property>
</bean>

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

The order of the `<bean/>` elements does not matter. The microcontainer orders the beans into the correct order. Which 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 Name2 is "Instantiated" (constructed) it is ok to configure it on Name1. Normally, injections wait for the referenced bean to reach the state "Installed".

```
public class Example{
    public Example getExample() {}
    public void setOther(Example other) {}
}

<bean name="Name1" class=com.acme.Example">
    <property name="number"><inject bean="Name2" state="Instantiated"/></property>
</bean>

<bean name="Name2" class=com.acme.Example">
```

```

    <property name="number"><inject bean="Name1"/></property>
</bean>

Example Name1 = new Example();
Example Name2 = new Example();
Name1.setOther(Name2); // We said we don't wait for a fully configured Name2
Name2.setOther(Name1); // Complete the configuration of Name2

```

Sometimes, we don't want to inject the bean itself. Instead, we want to inject some property of the bean. This is another form of factory, except the referenced bean is not really constructing the object.

```

public class Example{
    public String getHost() {}
    public void setHost(String host) {}
}

<bean name="URL" class="java.net.URL">
    <constructor><parameter>http://acme.com/index.html</parameter></constructor>
</bean>

<bean name="Name1" class=com.acme.Example">
    <property name="host"><inject bean="URL" property="host"/></property>
</bean>

URL url = new URL("http://acme.com/index.html");
Example example = new com.acme.Example();
example.setHost(url.getHost());

```

## 5.8. Collections

The `<collection/>`, `<list/>`, `<set/>` and `<array/>` all take the same form. Only `<list/>` is shown here.

The "elementClass" is required, unless you specify explicit types on all the `<value/>`s.

The default types are:

collection: java.util.ArrayList

list: java.util.ArrayList

set: java.util.HashSet

array: java.lang.Object[]

```

public class Example{
    public List getList() {}
    public void setList(List list) {}
}

<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 URL -->
        
```

```

        <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>

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`. For maps there are two default types for the elements, the `keyClass` and `valueClass`.

The `<entry/>` differentiates each `<key/>`, `<value/>` pair.

```

public class Example{
    public Map getMap() {}
    public void setMap(Map map) {}
}

<bean name="Name1" class=com.acme.Example">
    <property name="map">
        <map class="java.util.Hashtable" keyClass="java.lang.String" valueClass="java.lang.String">

            <entry><key>one</key><value>1</value></entry> <!-- uses keyClass and valueClass -->

            <entry><key class="java.lang.Integer">4</key><value>Four</value></entry> <!-- uses valueClass -->

            <entry><key>ten</key><value class="java.lang.Long">10</value></entry> <!-- uses keyClass -->

            <entry><key>bean</key><value><inject bean="SomeBean"/></value></entry>

        </map>
    </property>
</bean>

Example example = new com.acme.Example();
Map map = new Hashtable();
map.put(new String("one"), new String("1"));
map.put(new Integer(4), new String("4"));
map.put(new String("ten"), new Long(10));
map.put(new String("bean"), someBean);
element.setMap(map);

```

## 5.9. Lifecycle

Anybody familiar with the JBoss JMX microkernel will know about the lifecycle. The microcontainer extends the

lifecycle to all states of the POJO's:

#### Default Lifecycle:

Not Installed: The POJO has not been described or has been uninstalled.

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 occurred, usually due to misconfiguration.

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

```
<bean name="Name1" .../>
<bean name="Name2" ...>
  <depends>Name1</depends>
</bean>

// Deploy
name1.create();
name2.create();
name1.start();
name2.start();

// Undeploy
name2.stop();
name1.stop();
name2.destroy();
name1.destroy();
```

The "create", "start", "stop" and "destroy" methods can be overridden with parameters passed to them.

```
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>
```

## 5.10. ClassLoader

As of 1.0.2 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><null/></classloader>
</bean>
```

# 6

## Standalone

In this chapter we will look at how the standalone deployment works. The examples use the following class (at time of writing). You do not have to use this class, you trivially write your own class that uses the `BasicBootstrap` and `BeanXMLDeployer`.

```
/*
 * JBoss, the OpenSource J2EE webOS
 *
 * Distributable under LGPL license.
 * See terms of license at gnu.org.
 */
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.CollectionsFactory;

/**
 * Standalone Bootstrap of the kernel.
 *
 * @author <a href="mailto:adrian@jboss.com">Adrian Brock</a>
 * @author <a href="mailto:les.hazlewood@jboss.org">Les A. Hazlewood</a>
 * @version $Revision: 1.4 $
 */
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
    *
    * @param args the arguments
    * @throws Exception for any error
    */
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
 *
 * @param url the deployment url
 * @throws Throwable for any error
 */
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
 *
 * @param deployment the 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.

## Conclusion

The microcontainer is a powerful replacement for the JBoss's JMX microcontainer. 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.