BeanFactory Developer's Guide

Version 1.0 *Draft*

INTRODUCTION **3**

Overview 3
Purpose 3
Gaps in the JavaBean Component Model *3*
Gaps in the J2EE Architecture *3*
Design Objectives 4

ARCHITECTURE **5**

Named Components 5
Component Model 5
Container Model 5
Scope & Context 6
Declarative Configuration 6

INSTALLATION **8**

Without a J2EE Application Server 8
Installation *8*
Configuration *8*
With a J2EE Application Server 8
Installation *8*
Configuration *8*

BEANFACTORY CONTAINER REFERENCE **11**

Bean Names 11
Scope 11
Scope Context 13
APIs 13
Container Lookup APIs *13*
Services *14*
JavaBean Adapters *14*
Bean Definition Properties Files 15
File Format *15*
Special Properties *16*
Primitive Data Type Property Initializers *16*
Object Property Initializers *17*
Complex Data Type Property Initializers *17*
Bean Definition Properties Loading 18
Merging *18*
Properties Loading Sequence *19*
Bean Initialization Process 20
Container Initialization Process 20

MVC ARCHITECTURE REFERENCE **21**

BeanFactory HTTP Transaction Architecture 21
BeanFactory Taglibs 21
FormHandler Reference 21

Introduction

Overview

BeanFactory is a container for JavaBeans for non-GUI applications. It is a framework that allows software components to be developed using industry-standard design patterns and deployed as named resources within a container that provides lifecycle management. It allows applications to be built using a component-based building block approach that enables re-use, extensibility, maintainability and simplicity.

When used in conjunction with a J2EE application server BeanFactory offers a robust MVC (Model-View-Controller) framework on which web applications can be built. The MVC architecture builds upon the core container architecture in order provide a simple, elegant component model on top of which scalable and maintainable web applications can be built.

Purpose

The purpose of the BeanFactory is to fill a certain gaps in the J2SE and J2EE architectures.

Gaps in the JavaBean Component Model

The JavaBean component model is one of the most elegant component models ever devised. Although it was intended primarily to serve as the cornerstone of GUI applications, the basic design pattern is so simple, intuitive and useful that it is used in many non-GUI software applications. That said, it has several notable shortcomings:

1) There is no standard method to instantiate and initialize beans using the standard J2SE APIs.
2) There is no standard way to configure JavaBeans through external deployment descriptors.
3) JavaBean components do not have a well-defined lifecycle.
4) JavaBeans do not have a named service model ala JNDI.

The BeanFactory fills these gaps with a simple and unobtrusive container for JavaBeans.

Gaps in the J2EE Architecture

The J2EE specification provides a fantastic platform for modeling and implementing transactional software systems. Unfortunately, while it does a great job of solving some of the most complex problems, it is somewhat under-developed as an application framework. Although J2EE best practices allude to the use of a Model-View-Controller

framework, it is only in concept. There is no framework code that facilities an MVC architecture.

The lack of such an application framework tends to have disproportionately adverse effects on the extensibility, maintainability and cost of ownership for a given system. Even though application architects have the best of intentions, the lack of standardization leads to a lot of one-off architectures that become difficult and costly to implement.

The BeanFactory provides a consistent MVC architecture that enables web based applications to be built from JavaBean components that live within the BeanFactory container, which in turn lives within a J2EE application server.

Design Objectives

The BeanFactory architecture was designed such that it would be:

Component Oriented The BeanFactory leverages the industry standard JavaBean component model. It encourages re-use at both the class level and the component level.

Easy To Use The BeanFactory was designed in such a way, that it requires minimal knowledge to use. If you understand JavaBeans, you can use the BeanFactory.

Unobtrusive The BeanFactory was designed to do its job without getting in the way of the software architect and developer. It is a low-entropy architecture that does not dictate the use of custom object models or design patterns.

Lightweight The BeanFactory was designed to be equally useful in implementing large-scale enterprise applications as it is in implementing the proverbial HelloWorld application.

J2EE Compliant The BeanFactory is completely complementary to the J2EE platform. It builds on the J2EE platform rather than duplicates features already provided.

Open The BeanFactory is open source software. It leverages other high-quality open source software wherever possible.

Architecture

Named Components

The BeanFactory uses a named component model for all JavaBean components that it manages. The named component model allows components to be developed and plugged together much the same way that children assemble tinker toys. JavaBeans objects are located and instantiated using a simple URL naming syntax.

The following code is fairly self-explanatory.

```
Container.lookup("bean:/com/acme/MyBean");
```

It looks up a bean named /com/acme/MyBean, instantiates and initializes it if necessary and returns it to the caller. Depending on the declared scope of the bean, subsequent attempts to retrieve the same name would return the same bean.

Component Model

The BeanFactory architecture leverages one of the best component models on the market: JavaBeans. Rather than impose a custom object model in which components must extend and implement proprietary classes and interfaces, the BeanFactory only requires that one follow basic JavaBean conventions:

1) JavaBeans must have a public default or no-arg constructor.
2) JavaBeans must have getter and setter methods that conform to JavaBean naming conventions.

The design pattern could not be much simpler.

The BeanFactory improves on the existing JavaBean component model by providing a well-defined lifecycle for JavaBean components. Although EJBs have a very well defined lifecycle, the lifecycle of JavaBeans has been left largely undefined by Sun.

The BeanFactory addresses this lifecycle issue in several ways:

1) By providing a configuration subsystem that allows JavaBeans to be defined in a declarative manner using externalized deployment descriptors.
2) By providing a standard set of lifecycle callback methods.
3) By mandating that all BeanFactory-managed beans have a particular scope that determines the bounds of their existence.

Container Model

The BeanFactory borrows the concept of a "container" from the J2EE environment. The container provides the runtime services that manage the complete lifecycle of

BeanFactory-managed beans. It provides a layer of indirection between the component developer and the application assembler.

Scope & Context

In the BeanFactory architecture, a component's scope determines the context from which it can be obtained by URL. There is not a 1:1 relationship between URL and instance. Instead, the URL uniquely identifies an instance within a particular scope and calling context.

The three basic scopes that can be used to define a component's lifecycle are: static, thread and transient.

- Static scoped components have a lifespan that is consistent with the JVM itself. When the JVM shuts down, they can no longer be referenced by URL.
- Thread scoped components have a lifespan that is consistent with a particular calling thread. Retrieving the same URL from the same thread is guaranteed to return the same instance of the bean. Retrieving the same URL from different threads is guaranteed to return a different instance of the bean.
- Transient scoped beans have no lifecycle at all. Subsequent calls to retrieve a particular URL are always guaranteed to return a different JavaBean instance.

One of the objectives of the BeanFactory architecture is to ensure that it leverages the J2EE architecture wherever possible. The J2EE servlet architecture has three well-defined scopes for web applications: application, session, and request.

- Application scoped components have a lifespan that is consistent with a web application.
- Session scoped components have a lifespan that is consistent with the http session.
- Request scoped components have a lifespan that is no longer than a single http transaction.

When used in conjunction with a J2EE application server, the BeanFactory leverages the J2EE servlet container to maintain instances within the context of these three scopes.

Declarative Configuration

One of the key design goals of the BeanFactory architecture was to have a purely declarative configuration system that would enable entire in-memory structures to be defined outside of Java code. The ability to declare entire object graphs has certain advantages:

1) Architectural transparency. While OO designs are often quite elegant on paper, by the time they are implemented they can be quite opaque are difficult to penetrate. The BeanFactory provides architectural transparency down to the deepest internals of an application.

2) Consistency. The BeanFactory minimizes the need for ad-hoc design patterns. While OO design patterns are very useful, they have a way encouraging architects and developers to be too clever for their own good. This is usually manifest by object factories scattered about with configuration files of various incompatible formats. The BeanFactory is a combines two very simple patterns: JavaBeans and the factory pattern.
3) Configurability. A powerful declarative configuration systems allows applications to configured and reconfigured over time with minimal need to examine, change or re-compile code.
4) Extensibility. The combination of architectural transparency and configurability ensures that a system is extensible. If one wants to change the behavior of a core system, most often the only option is to edit the source code and recompile the application. By externalizing declaration of components and their implementing classes, it is possible to enhance behavior of system internals, simply by extending a given class, implementing new functionality and reconfiguring the system to use the extended class. Without an externalized declarative configuration system this is difficult or impossible.

Installation

Without a J2EE Application Server

Installation

Installation is trivial. Just put the "beanfactory.jar" file in your class path, preferably near the end. The reason for placing it at the end of your class path will be explained later.

The following shows how this might be done:

```
java -cp myclasspath:beanfactory.jar mypackage.MyClass
```

Configuration

For now, the simplest way to configure BeanFactory-managed beans is to place a file named 'beanfactory.properties' somewhere in the root of your class path. When the container initializes, it will search for all such files in the class path and will source them appropriately.

For now this file can be empty.

As you will see in later sections, there are many different ways to configure beans. Using beanfactory.properties is just a simple way to get started.

With a J2EE Application Server

Installation

Installation in a web application environment is simple. Just place beanfactory.jar and beanfactory-servlet.jar in the WEB-INF/lib directory of your web application. You should **not** place beanfactory.jar in any other class path for your application server.

However, certain application servers have web application class loaders that have broken implementations of getResource(). Because the BeanFactory uses this method to bootstrap itself, it may be necessary to place beanfactory.jar outside of WEB-INF/lib and include it in an appropriate class path. Please refer to your application server's documentation for the correct place to put this file. It will depend on the class loading mechanism used by your particular application server. In any case, beanfactory-servlet.jar should always be loaded out of WEB-INF/lib. Failing to place beanfactory-servlet.jar in the WEB-INF/lib directory of your web application will cause the MVC subsystem to fail as well.

Configuration

Copy the bftaglib.tld taglib descriptor to your WEB-INF directory. If you are using a Servlet 2.2 compliant application server, copy the web22.xml to your WEB-INF directory

and rename it to 'web.xml'. If you are using a Servlet 2.3 compliant application server, copy the web.xml file in the BeanFactory distribution to your WEB-INF directory. We recommend that you use a servlet 2.3 compliant application server such as WebLogic 6.1 or Tomcat 4.x.

If you intend to use the BeanFactory with an already-configured web application, you will need to copy and paste the various sections into your existing web.xml file. There is nothing tricky about this, but please remember that according to the web.xml DTD, order of the declarations does matter

BeanFactory Container Reference

Bean Names

All java beans that are loaded by the BeanFactory container have a unique name.

BeanFactory bean names contain a name for the Bean preceded by a namespace. Examples of such names are:

/MyBean
/my/package/MyBean

By convention, these names follow capitalization conventions of fully qualified java classes. They can, but need not, match the fully qualified class name of the declared java class. That is, for a given class, my.package.MyBean, the bean name could be /my/package/MyBean, but it could just as well be /some/other/naming/scheme/FooBean.

If, for a given bean class, it is likely that there would be only instance, the fully qualified class name would most likely be used. If, on the other hand, the bean class could have many different instances, the naming is not likely to match.

Scope

Beans that are managed by the BeanFactory container all contain a declared scope. The valid scopes are: static, thread, transient, application, session and request. The first three are valid for any application. The latter three are valid only within a J2EE web application.

Scope	Summary	Identity
Static	Static scoped beans are "global" with respect to the JVM. (More accurately, they are global with respect to the Class loader used to load the container.)	Two lookups for the same bean name will return a reference to the same instance if and only if they are called from the same thread.
Thread	Thread scoped beans are resolvable relative to the execution of a given thread. If a thread dies, all instantiated beans within that scope will be unreachable and eligible for garbage collection.	Two lookups to the same bean will return the same reference if and only if they are called from the same thread.
Transient	Transient beans do not have a real scope. After being	Two calls to look up the same bean name will never

	instantiated, the container does not keep a reference to them. This makes transient roughly the same as a normal Java factory pattern.	return the same instance.
Application	Application scope bean instances are maintained by the underlying J2EE servlet container. Internally the beans are stored by using the getAttribute()/setAttribute() methods of Servlet Context. They have the same lifecycle as a ServletContext instance.	Two calls to look up an application scoped bean will return the same instance if and only if both lookups are called from within the same web application.
Session	Application scoped bean instances are maintained by the underlying J2EE servlet container. Internally they are stored by using the getAttribute()/setAttribute() methods of HttpSession. They have the same lifecycle as an HttpSession instance.	Two calls to look up an application scoped bean will return the same instance if and only if both lookups are called from within the same HttpSession.
Request	Request scoped beans are maintained by the underlying J2EE servlet container. Internally they are stored using the getAttribute/setAttribute methods of HttpServletRequest. They have the same lifecycle as an HttpServletRequest instance.	Two calls to look up a request scoped bean will return the same instance if and only if both lookup operations are called from the same HttpRequest.

The key concept to understand here is that the bean name-to-instance mapping is not a 1:1 relationship except in the case of 'static' beans. A bean's name can only be resolved with respect to its declared scope. For instance, if a bean is declared as a request scoped component, it can have as many instances as there are HttpRequests. *A particular instance must be resolved by its name and its scope*. Neither name nor scope alone is sufficient to resolve a bean.

At this point you might be asking, "How does the container know which application , session or request context it should use to resolve a particular URL?"

The answer is that all operations against the BeanFactory container occur in a particular scope context that is bound to the currently executing thread.

Scope Context

Before the BeanFactory can resolve beans of thread, application, request or session scope, the calling thread must be registered with the BeanFactory. In a web application this is done automatically by a servlet filter in a servlet context.

This makes it possible to resolve beans without having to keep explicitly keep references to their containers.

In fact, there are a few special "magic" URLs that are mapped in any Servlet environment. `/javax/servlet/ServletContext`, `/javax/servlet/http/HttpRequest` and `/javax/servlet/HttpResponse` can be used to resolve the respective `ServletContext`, `HttpRequest` and `HttpResponse` objects.

Because beans can resolve these objects anywhere within a bound context, it is possible to write code that is much cleaner. That is, developers do not need to ensure that these objects are passed throughout the call stack. This added level of indirection allows tremendous flexibility with regard to building clean and modular object models. It enables applications to be architected with certain assumptions that establish a contract about the execution environment.

In this regard, the BeanFactory provides a similar environment to an EJB container, but with lighter weight components that are optimized for front-end application development.

APIs

The full BeanFactory JavaDoc API reference can be found online at this location: http://www.beanfactory.net/javadoc/index.html However, browsing the JavaDoc belies the simplicity of actually using the Framework, so we recommend that you read this API summary first.

Container Lookup APIs

The BeanFactory Container APIs are very simple to use. They all exist in the `gnu.beanfactory.*` package. There is a single class `gnu.beanfactory.Container` which provides access to 90% of the functionally that one needs to use the BeanFactory. It has two static methods that are nearly identical:

```
java.lang.Object lookup(String beanURL) throws BeanFactoryException
java.lang.Object resolve(String beanURL) throws BeanFactoryException
```

Both of these methods take beanURLs and return objects from the container. The `resolve()` method implements a superset of the functionality that `lookup()` implements. The only difference between the two is that `resolve()` will resolve nested properties while `lookup()` will not.

The following code snippits give examples of how these methods would be used:

```
Foo foo = (Foo) Container.lookup("bean:/Foo");

Bar bar = (Bar) Container.resolve("bean:/Foo.bar");
```

In the first, we look up a bean named `/Foo`. In the second example, we look up the bar property of the `/Foo` bean. This is functionally equivalent to calling `foo.getBar();`

Services

Although any java class with a default constructor can be used with the BeanFactory, there is a special class that implements some useful behavior. The gnu.beanfactory.Service class allows one to implement "services" that run in their own threads. The Service class implements java.langRunnable, so all one needs to do is implement the run() method. By setting the $startup property, these services can be set to start automatically when the Container initializes.

JavaBean Adapters

There are often APIs which implement their own factory patterns that would prevent them from being used. The JDBC API is a good example. Connection instances are created via the new operator, but rather by the DriverManager class

One can use these APIs with the by using the gnu.beanfactory.FactoryAdapter interface. It declares a single method Object createBean(). If the BeanFactory sees that a class that it is instantiating implements this interface, it will invoke createBean() and return that value.

In the following pseudocode, we demonstrate the pattern:

```
public class ConnectionAdapter implements
gnu.beanfactory.FactoryAdapter {

	String myURL;

	public String getURL() {
		return myURL;
	}
	public void setURL(String url) {
		myURL = url;
	}
	public Object createBean() {
		return DriverManager.getConnection(url,"user","pass");
	}
}
```

Using the following properties,

```
/MyConnection.$class = MyAdapter
/MyConnection.url=jdbc:url:goes:here
```

we can use the following code to look up a Connection as if it was a bean.

Connection myConnection = (Connection) Container.lookup("bean:/MyConnection");

It is worth noting that objects that are exposed through the FactoryAdapter interface obey all of the scope rules of normal JavaBeans within the container.

Bean Definition Properties Files

The BeanFactory uses standard properties files to declare and define the JavaBean components that are managed by the BeanFactory container. Although properties files are less powerful than XML-based configuration formats, they have two key advantages:

1) Properties files are less verbose and easier to read than XML
2) Properties files can be layered and merged on top of one and other with ease.

The latter of these two advantages is essential to building a framework that yields highly configurable applications.

File Format

The basic syntax for bean definition files is the following:

[*fully qualified bean name*].[*property*]=[value]

A *fully qualified bean name* is similar in concept to a fully qualified class name in Java. The key differences are that fully qualified bean names a) refer to instances, not classes and b) are separated by slashes '/' instead of dots '.'. Examples of fully qualified bean names are: `'/my/namespace/MyBean'` and `'/AnotherBean'`.

The *property* portion specifies either a JavaBean property name to be initialized or a special property that is used by the BeanFactory container. If it is a JavaBean property, it should adhere to standard JavaBean naming conventions. (See next section for explanation of special properties.

The *value* portion of a property definition represents either a value to which the JavaBean property value should be initialized or a URL reference to another JavaBean.

The following set of properties declares a bean named /my/Bean to have static scope.

/my/Bean.$class=my.BeanClass

/my/Bean.$scope=static

Special Properties

There are a number of special properties that are used to define a bean instance. All of these special properties begin with a dollar sign '$'. They are:

Special Property	Required	Description	Valid Values
`$class`	Required	Specifies the fully qualified class name for the bean.	Any fully qualified class name.
`$scope`	Optional	Specifies the scope of the bean in the container. If this property is not specified, the value of 'static' is implicit.	`static`, `transient`, `thread`, `request`, `session` or `application`
`$startup`	Optional	Specifies whether or not the bean should be initialized when the container is initialized.	`true` or `false`
`$parent`	Optional	The fully qualified URL of a bean from which to inherit properties.	A fully qualified bean URL of the form `bean:[bean name].` Example: `bean:/my/Bean`
`$delimiter`	Optional	Specifies the delimiters for multi-valued attributes such as arrays, Vectors and Hashtables. Defaults to ",l".	A string containing valid delimiter characters.

Primitive Data Type Property Initializers

All primitive data types can be initialized by the BeanFactory container. Primitive wrapper objects can be initialized as well. The complete list is:

```
boolean, byte, char, int, long, float, double, java.lang.String,
java.lang.Byte, java.lang.Character, java.lang.Integer,
java.lang.,Long, java.lang.Float, java.lang.Double,
java.math.BigInteger, java.math.BigDecimal
```

String values from the properties definition are automatically converted into the data type declared by the JavaBean property.

Object Property Initializers

One of the most powerful features of the BeanFactory is its ability to configure entire object graphs in a declarative fashion. That is, any JavaBean property that holds an object reference can be initialized by the container. This can be accomplished by specifiying a fully qualified beanfactory URL. In the following example, /my/Bean will be initialized with a reference to /another/Bean

/my/Bean.myProperty=bean:/another/Bean

The BeanFactory container is can resolve circular references without a problem.

Complex Data Type Property Initializers

The BeanFactory has first-class support for four kinds of complex data types: arrays, `java.util.Vector`, `java.util.Hashtable`, `java.util.Properties`, Providing first class support for these data types makes it possible to initialize very complex object graphs using only a declarative syntax.

The values of arrays can be: primitives, primitive wrappers or Objects:

```
# Initialize an array of int primitvies

MyBean.intArray=1,2,3,4

# Initialize an array of objects using URL references

MyBean.objectArray=bean:/SomeBean,bean:/AnotherBean
```

Vectors behave like arrays of java.lang.Object. The values will be interpreted as Strings unless the values begin with bean:, in which case they will be resolved by URL.

```
# The following will initialize a vector
# containing three strings.

/MyBean.myVector = a,b,c

# The following will initialize a vector containing two beans:
/MyBean.myVector=bean:/Foo, bean:/Bar

# The following will initialize a vector
# containing one bean and one string:
/MyBean.myVector=bean:/Foo, SomeStringValue
```

Hashtables obey the same rules as Vectors, albeit with a slightly different syntax:

```
# A Hashtable of String->String mappings ("a"->"1", "b"->"2")
/MyBean.myHashtable =a=1,b=2

# Hashtable keys and/or values can contain URL references
# 1) A bean /A will be mapped to the string "A"
```

```
# 2) The String "B" will be mapped to the bean /B
# 3) The bean /Foo will be  mapped to the bean /Bar
/MyBean.myHashtable=bean:/A=A,B=bean:/B,bean:/Foo=bean:/Bar
```

Properties are handled slightly differently from Hashtables in that URL references are not resolved.

```
# Will contain a mapping of strings: "bean:/A" -> "bean:/B"
/MyBean.myProperties=bean:/A=bean:/B
```

Bean Definition Properties Loading

Merging

The BeanFactory container is initialized by one large master set properties. However these properties are loaded from many separate properties files and merged one on top of another.

Consider an example in which there are two properties files, config1.properties and config2.properties. Config1.properties is loaded first and then config2.properties is loaded. The contents of these files are as follows

Config1.properties:

```
/A.$class=a.b.c
/A.someProperty=abc
```

Config2.properties:

```
/B.$class=d.e.f
/A.someProperty=override
```

The effective merged property set would be:

```
/A.$class=a.b.c
/A.someProperty=override
/B.$class=d.e.f
```

You will note that the initialization value of "someProperty" is overridden by the value in config2.properties.

Arrays, Vectors, Hashtable and Properties have a special syntax that allows the values to be appended together. Rather than replace the values that already exist, the following would append
If config1.properties contains:

/A.arrayProperty=1,2,3

and config2.properties contains:

/A.arrayProperty+=4,5

Then the effective merged properties would contain:

/A.arrayProperty=1,2,3,4,5,6

Properties Loading Sequence

The way in which properties are loaded is slightly different depending on whether the loading is taking place in a J2EE Web Application Context or not. The following will take place regardless of whether the Container is being loaded in a WebApplication context or not.

1) The class loader that loaded the Beanfactory.jar, will search its class path for all resources named "/beanfactory.properties". These beanfactory.properties files will be sourced from right to left in the class path. That is, if the class path is JarA.jar;JarB.jar, and both jars contain a beanfactory.properties file, the one in JarB.jar will be sourced first. This is consistent with the left-to-right precedence of the CLASSPATH concept. That is, properties in JarA.jar will take precedence over properties in JarB.jar.
2) After the class path has been searched, the container will look for a system property class beanfactory.configpath (specified via –Dbeanfactory.configpath when the JVM is invoked). The configpath will contain a semicolon ';' delimited list of properties *files* that will be sourced from right to left and merged onto the properties that were loaded from the class path. Again the leftmost properties files will take precedence over files to the right.

This will establish the base set of properties. For each web application that is loaded this base set of properties will be used and properties will be merged on top using the following sequence:

1) The Servlet Container will search for "`/WEB-INF/beanfactory.properties`" If it exists, it will be sourced and merged on top of existing properties.
2) The Servlet Container will check to see if the web application property "`beanfactory.configpath`" has been set. If it is set, it will be interpreted as a semicolon delimited set of properties files that will be parsed and merged from right to left (again, consistent with standard CLASSPATH precedence). These resources are relative to the web application root, so an example `beanfactory.configpath` might be "`/WEB-INF/foo.properties;/WEB-INF/bar.properties`"

Please note that ***static scoped beans should not be declared from a properties file sourced by the Servlet Context.*** If the container detects this, it will reject and unload the bean definition. This helps to ensure that web applications do not interfere with each other.

Bean Initialization Process

The bean initialization process looks something like this:

1) `Container.lookup()` is called to obtain a bean by its url.
2) The Container looks up the declared scope for the bean
3) The container looks inside the proper scope context to see if an instance already exists. If it exists, it is returned to the user.
4) If the bean does not exist, the container instantiates the proper java bean class. The default constructor is invoked.
5) The container checks to see if there is a method `preInit()`. If it exists, it is invoked.
6) The container initializes all JavaBean properties that are specified in the bean definition. If URL references must be resolved, the container will walk through the object graph and initialize all dependent beans.
7) The container checks to see if there is a method `postInit()`. If it exists, it is invoked.
8) The bean placed in the proper scope context.
9) The bean is returned to the caller.

Container Initialization Process

The BeanFactory container is initialized automatically upon first use. There is no explicit initialization API that one must call. Any BeanFactory API call will cause the container to initialize itself. The BeanFactory will send a fair amount of information to the console regarding the initialization process. (This output is generated through Log4J, so it can be turned off and/or redirected to another location.)

When the container initializes, it follows the following process:

1) The container loads a master set of bean definition properties. It does this according to the process outlined in the previous section.
2) For each declared bean, the BeanFactory validates that the bean definition is valid. Invalid bean definitions are removed from the system and an appropriate error message is logged.
3) The container iterates though all bean definitions that have the '$startup' property set to true. These are loaded in sequence and placed in the proper scope context. If these beans throw an exception during initialization they are unloaded from the system, an error message is logged and initialization continues.
4) The container proceeds with the user-initiated action that caused it to initialize itself, most likely and invocation of Container.lookup().

If a web application is being initialized that uses the BeanFactory, some secondary initialization steps are taken.

1) A secondary properties set is sourced by the web application context loader.
2) Bean definitions declared by the web application are validated. Any Bean Definitions declared within a servlet context that are declared to have 'static'

scope are unloaded by the system. This is done to prevent collisions between multiple web applications.

3) All beans declared with the $startup property set to true are initialized.

MVC Architecture Reference

BeanFactory HTTP Transaction Architecture

TBD

BeanFactory Taglibs

TBD

FormHandler Reference

TBD