Snowdrop 1.0

User Guide

by Marius Bogoevici and Aleš Justin

What This Guide Covers v
1. Introduction 1
1.1. Structure of the package 1
2. Component usage 3
2.1. The VFS-supporting application contexts 3
2.2. Load-time weaving 5
2.3. The Spring Deployer 5
2.3.1. JBoss + Spring + EJB 3.0 Integration 5
2.3.2. Installation 6
2.3.3. Spring deployments 6
2.3.4. Deployment 6
2.3.5. Defining the JNDI name 6
2.3.6. Parent Bean factories 7
2.3.7. Injection into EJBs 7

What This Guide Covers

Snowdrop is a utility package that contains JBoss-specific extensions to the Spring Framework. These extensions are either:

- Extensions to Spring Framework classes, that can be used wherever the generic implementations provided by the framework do not integrate correctly with the JBoss Application server
- JBoss AS-specific extensions for deploying and running Spring applications

This user guide aims to cover the functionality of Snowdrop, to describe its components, and to provide information on how to use it optimally for running Spring applications in JBoss.

The current version of the package is supporting the following configuration:

- JBoss AS 5.x
- Spring 2.5.x

Introduction

1.1. Structure of the package

Snowdrop contains libraries that offer support for:

- resource scanning - i.e. scanning the classpath for bean definitions or using "classpath*:"-style patterns
- load-time weaving
- bootstrapping and registering application contexts to be used by Java EE applications

The distribution contains the following files:

- snowdrop-vfs.jar - the library containing the support classes for resource scanning
- snowdrop-weaving.jar - the library containing the support classes for load-time weaving
- spring-deployer.zip - the Spring Deployer

Component usage

This chapter details how to use the individual components of the package.

2.1. The VFS-supporting application contexts

For using this functionality, the snowdrop-vfs.jar file needs to be added to the application.

This library supports resource scanning in JBoss' Virtual File System. When doing resource scanning, the Spring framework assumes that the resources are either coming from a directory or a packaged jar, and treats the URLs encountered in the process of scanning resources accordingly. This assumption does not hold in JBoss' Virtual File System.

The solution to this problem is in implementing a different underlying resource resolution mechanism, namely in amending the functionality of the `PathMatchingResourcePatternResolver`. From the user's perspective, using this different resolution mechanism is done through using one of the two ApplicationContext implementations provided by this library, which are:

- `org.jboss.spring.vfs.context.VFSClassPathXmlApplicationContext` - to be used instead of `org.springframework.context.support.ClassPathXmlApplicationContext`
- `org.jboss.spring.vfs.context.VFSXmlWebApplicationContext` - to be used instead of `org.springframework.web.context.support.XmlWebApplicationContext`

In many cases, the `VFSClassPathXmlApplicationContext` would be instantiated on its own, using something like:

```
ApplicationContext context =
  new VFSClassPathXmlApplicationContext("classpath:/context-definition-file.xml");
```

The XmlWebApplicationContext is not instantiated directly, though, but bootstrapped by either the ContextLoaderListener, or the DispatcherServlet. In this case, the class to be used for bootstrapping needs to be used to trigger an instantiation of the VFS-enabled context.

For changing the type of application context created by the ContextLoaderListener, use the contextClass parameter, as shown in the sample below:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath*:spring-contexts/*.xml</param-value>
</context-param>
<context-param>
  <param-name>contextClass</param-name>
  <param-value>
    org.jboss.spring.vfs.context.VFSXmlWebApplicationContext
  </param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

For changing the type of application context created by the DispatcherServlet, use the contextClass parameter again, but this time on the `DispatcherServlet` definition (emphasized portion again):

```
<servlet>
  <servlet-name>spring-mvc-servlet</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/mvc-config.xml</param-value>
  </init-param>
  <init-param>
  <param-name>contextClass</param-name>
    <param-value>
      org.jboss.spring.vfs.context.VFSXmlWebApplicationContext
    </param-value>
  </init-param>
</servlet>
```

Both configurations can be seen at work in the web-scanning sample.

Note

In general, it is a good idea to pay attention to this error. If encountered while the application is starting, you definitely need to replace the default ApplicationContext with one of the VFS-enabled implementations.

```
Caused by: java.util.zip.ZipException: error in opening zip file
  ...
  at org.springframework.core.io.support.PathMatchingResourcePatternResolver
 .doFindPathMatchingJarResources(PathMatchingResourcePatternResolver.java:448)
```

(the listing has been wrapped for formatting purposes).

2.2. Load-time weaving

In order to perform load-time weaving for the application classes in Spring (either for using load-time support for AspectJ or for JPA support), the Spring framework needs to install its own transformers in the classloader. In certain cases (like for JBoss 5.x), a classloader-specific LoadTimeWeaver is necessary. The functionalities described in this chapter are included in the snowdrop-weaving.jar file.

To that effect, if a load-time weaver needs to be defined in the www Spring application context, use the JBoss5LoadTimeWeaver class, as follows:

```
<context:load-time-weaver                                          weaver-
class="org.jboss.instrument.classloading.JBoss5LoadTimeWeaver"/>
```

2.3. The Spring Deployer

The role of the Spring Deployer is to allow the bootstrapping of a Spring application context, binding it in JNDI and using it for providing Spring-configured business object instances.

2.3.1. JBoss + Spring + EJB 3.0 Integration

This distribution contains a JBoss Deployer that supports Spring packaging in JBoss. What this means is that you can create JAR archives with a *META-INF/jboss-spring.xml* file and your Spring bean factories will automatically be deployed. Also supported in this distribution is EJB 3.0 integration. You can deploy Spring archives and be able to inject beans created in these deployment directly into an EJB using a @Spring annotation.

2.3.2. Installation

If you are using EJB 3.0 and JDK 5 integration, copy the jboss-spring-jdk5.deployer directory into the JBoss deploy/ directory. If you are using JDK 1.4, then copy the jboss-spring.deployer/ into the deploy directory. If you look inside these *.deployer* deployments you will see that only a partial Spring distribution is contained. If you need a full Spring distribution, then copy those jars into the *.deployer* directory or into the lib/ directory of your JBoss configuration.

2.3.3. Spring deployments

You can create Spring deployments that work much in the same way .sar's, .war's, .ear's, .har's, and .rar's work. Using the JBoss Spring deployer you can create Spring jars:

```
my-app.jar/
   org/
      acme/
          MyBean.class
          MyBean2.class
   META-INF/
           jboss-spring.xml
```

So, my-app.JAR is a jar that contains classes, like any other JAR and a jboss-spring.xml file in the META-INF/ of the jar. This jboss-spring.xml file is like any other Spring xml file. By default, the JBoss Spring Deployer will register this bean factory defined in the XML file into JNDI. It will be registered in a non-serialized form so you don't have to worry about JNDI serialization! The default JNDI name will be the short name of the deployment file. So in this example, the bean factory described in the *META-INF/jboss-spring.xml* file will be registered under the "my-app" JNDI name.

Alternatively, you do not have to create an archive. You can put your jar libraries under server/<config-name>/lib and just put an XML file of the form: <name>-spring.xml into the JBoss deploy directory. For example, my-app-spring.xml. Again, the JNDI name will be by default, the short name of the XML file, in the case my-app-spring.xml will produce a JNDI binding of "my-app".

2.3.4. Deployment

Once you have created a *.spring* archive or a *-spring.xml* file, all you have to do is put it in the JBoss deploy/ directory and it will be deployed into the JBoss runtime. You can also embed these deployments inside an EAR, EJB-JAR, SAR, etc. as JBoss supports nested archives.

2.3.5. Defining the JNDI name

You can specify the JNDI name explicitly by putting it in the description element of the Spring XML.

```
<beans>
   <description>BeanFactory=(MyApp)</description>
   ...
   <bean id="springBean" class="example.SpringBean"/>
</beans>
```

MyApp will be used as the JNDI name in this example.

2.3.6. Parent Bean factories

Sometimes you want your deployed Spring bean factory to be able to reference beans deployed in another Spring deployment. You can do this by declaring a parent bean factory in the description element in the Spring XML.

```
<beans>
  <description>BeanFactory=(AnotherApp) ParentBeanFactory=(MyApp)</description>
  ...
  </beans>
```

2.3.7. Injection into EJBs

Once an ApplicationContext has been successfully bootstrapped, the Spring beans defined in it can be used for injection into EJBs. To that end, the EJBs must be intercepted with the SpringLifecycleInterceptor, as in the following example:

```
@Stateless
@Interceptors(SpringLifecycleInterceptor.class)
public class InjectedEjbImpl implements InjectedEjb
{
   @Spring(bean = "springBean", jndiName = "MyApp")
   private SpringBean springBean;

   /* rest of the class definition ommitted */
}
```

In this example, the EJB InjectedEjbImpl will be injected with the bean named 'springBean', defined in the ApplicationContext previously described in sections 3.3 and 3.5.