Spring

j2ee Application Framework

Spring Framework Reference Documentation

Version: 1.0

Table of Contents

Chapter 1. Introduction

1.1. Lightweight containers

There's a lot of interest in what we call lightweight containers these days. We see this as the future, especially where web applications are concerned, but also for reuse of e.g. business and data access objects in both J2EE environments and standalone applications.

What is a lightweight container? EJB is perhaps the best counterexample:

1. Invasive API (your code depends on EJB)
2. Container dependency (your code won't work outside of an EJB container)
3. Fixed set of capabilities that can't be configured
4. edicated deployment steps
5. Long startup time
6. Dedicated deployment steps

Lightweigth containers aim to avoid all the above inconveniences.

1.2. Some more blabbering

Here goes some more blabbering about Spring!

Chapter 2. High level Overview

High level overview of Spring.

Chapter 3. The Bean Package

3.1. Introduction

Spring's core is the `org.springframework.beans` package, designed for working with JavaBeans, retrieving objects by name and managing relationships between objects. The beans package and its subpackages provide functionality for specifying the infrastructure of a project using JavaBeans.

The most important concept in the beans package is the `BeanFactory`. The BeanFactory is a generic factory, able to instantiate objects and manage relationships between different objects. Objects managed by the BeanFactory can be of several types and support is provided for e.g. initialization methods. Also, there's a lot callback interfaces you can implement to get the right behavior.

The rest of this chapter discusses the different concept of the beans package, like the BeanFactory, the different callback interfaces available.

3.2. Basic bean functionality

The `org.springframework.beans` package adheres to the JavaBeans standard provided by Sun. A JavaBean is simply a class with a default no-argument constructor, which follows a naming conventions where a property named `prop` has a setter `setProp(...)` and a getter `getProp()`. For more information about JavaBeans and the specification, please refer to Sun website (java.sun.com).

The `org.springframework.beans` package provides the ability to manipulate beans easily and adds enhanced functionality like standard methods to perform initialization after properties have been set and other lifecycle features.

The basis for the beans package is the `BeanWrapper` interface and its corresponding implementation (`BeanWrapperImpl`). As quoted from the JavaDoc, the BeanWrapper offers functionality to set and get property values (individually or in bulk), get property descriptors and query the readability and writability of properties. Also, the BeanWrapper offers support for nested properties, enabling the setting of properties on subproperties to an unlimited depth. Last but not least, the BeanWrapper support the ability to add standard JavaBeans `PropertyChangeListeners` and `VetoableChangeListeners`, without the need for supporting code in the target class. The BeanWrapper usually isn't used by application code directly, but by the `DataBinder` and the `BeanFactory`.

3.3. The BeanFactory

The `org.springframework.beans.factory` package and its subpackages provide the basis for Spring IoC (Inversion of Control, see TODO REFERENCE). The `BeanFactory` provides functionality for managing objects and their relationships and retrieving their respective instance(s).

3.3.1. To singleton or not to singleton

Beans exist in two types, singletons and prototypes. When a bean is a singleton, only one *shared* instance of the bean will be managed and all requests for instances of that specific instances of bean will result in that one specific bean instance being returned.

The prototype mode of a bean results in *creation of a new bean instance* every time a request for that specific bean is being done. This is ideal for situation where for example each user needs an independent user object or something similar.

3.3.2. Clients interacting with the factory

The client-side view of the BeanFactory is surprisingly simple. The `BeanFactory` interface four methods for clients to interact with it:

- `Object getBean(String)`: returns an instance of the bean registered under the given name. Depending on how the bean was configured by the beanfactory configuration, a singleton and thus shared instance will be returned, or a newly created bean. A `BeansException` will be thrown when either the bean could not be found (in which it'll be a `NoSuchBeanDefinitionException`), or an exception occured while instantiated and preparing the bean
- `Object getBean(String,Class)`: returns a bean, registered under the given name. The bean returned will be cast to the given Class. If the bean could not be cast, corresponding exceptions will be thrown (`BeanNotOfRequiredTypeException`). Furthermore, all rules of the getBean(String) method apply (see above)
- `boolean isSingleton(String)`: determines whether or not the beandefinition registered under the given name is a singleton or a prototype. If the beandefinition could corresponding the given name could not ben found, and exception will be thrown (`NoSuchBeanDefinitionException`)
- `String[] getAliases(String)`: returns the aliases configured for this bean (TODO: WHAT IS THIS :)

3.3.3. BeanFactory implementations

Some explanation on the different implementations of the beanfactory

3.4. Bean Lifecycle Methods

Spring provides a couple of marker interfaces and some other features that allow customizing the lifecycle of a bean managed by a beanfactory. Each of the marker interfaces as well as the other features and the the functionality they offer are described below.

3.4.1. FactoryBean

The `org.springframework.beans.factory.FactoryBean` is to be implemented objects that *are themselves factories*. The BeanFactory interface provides three method:

- `Object getObject()`: has to return an instance of the object this factory creates. The instance can possibly be shared (depending on this factory providing singleton or prototypes).
- `boolean isSingleton()`: has to return `true` if this FactoryBean returns singletons, `false` otherwise
- `Class getObjectType()`: has to return either the object type returned by the `getObject()` method or `null` if the type isn't known in advance

3.4.2. InitializingBean

The `org.springframework.beans.factory.InitializingBean` gives you the ability the do initialization work after all necessary properties on a bean are set by the BeanFactory. The InitializingBean interface specifies exactly one method:

- `void afterPropertiesSet()`: called after all properties have been set by the beanfactory. This method enables you to do checking if all necessary properties have been set correctly, or to perform initialization work. You can throw *any* exception to indicate misconfiguration, initialization failures, etcetera

Note: generally, the use of the `InitializingBean` can be avoided (and by some people is discouraged). The beans package provides support for a generic init-method, given to the beandefinition in the beanconfiguration store (may it be XML, properties-files or a database). For more information about this feature, see the next section)

3.4.3. init-method

Besides the InitializingBean, Spring also offers a less intrusive way of defining an initialization method on your beans. Each of the implementations of the BeanFactory specifies this features in a different way, but they all have the same result: a no-argument method getting called after all properties have been set and *also after the `afterPropertiesSet()` method from InitializingBean has been called*. TODO INCLUDE REFERENCE TO DIFFERENT WAYS TO SPECIFY INIT_METHODS????

3.4.4. DisposableBean

The `org.springframework.beans.factory.DisposableBean` interface provides you with the ability to get a callback when a beanfactor is destroyed. The DisposableBean interface specifies one method:

- `void destroy()`: called on destruction of the beanfactory. This allows you to release any resources you are keeping in this bean (like database connections). You can throw an exception here, however, it will not stop the destruction of the bean factory. It will get logged though.

Note: generally, the use of the `DisposableBean` can be avoided (and by some people is discouraged). The beans package provides support for a generic destroy-method, given to the beandefinition in the beanconfiguration store (may it be XML, properties-files or a database). For more information about this feature, see the next section)

3.4.5. destroy-method

Besides the Disposable, Spring also offers a less intrusive way of defining an destroy methods on your beans. Each of the implementations of the BeanFactory specifies this features in a different way, but they all have the same result: a no-argument method getting called after all properties have been set and *also after the `destroy()` method from DisposableBean has been called*. TODO INCLUDE REFERENCE TO DIFFERENT WAYS TO SPECIFY DESTROY_METHODS????

3.4.6. BeanFactoryAware

The `org.springframework.beans.factory.BeanFactoryAware` interface gives you the ability to get a reference to the BeanFactory that manages the bean that implements the BeanFactoryAware interface. This feature allows for implementing beans to look up their collaborators in the beanfactory. The interface specifies one method:

- `void setBeanFactory(BeanFactory)`: method that will be called *after the initialization methods* (`afterPropertiesSet` and the init-method).

Chapter 4. The Context Package

Chapter 5. Sending Email

Chapter 6. Validating data objects

Chapter 7. Aspect Oriented Programming with Spring

Chapter 8. Sourcelevel Metadata Support

Chapter 9. Data Access using JDBC

Chapter 10. Data Access using O/R Mappers

Chapter 11. Web framework

11.1. Introduction to the web framework

Spring's web framework is designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, and locale and theme resolution. The default handler is a very simple Controller interface, just offering a "ModelAndView handleRequest(request,response)" method. This can already be used for application controllers, but you will prefer the included implementation hierarchy, consisting of AbstractController, AbstractCommandController, MultiActionController, SimpleFormController, AbstractWizardFormController. Application controllers will typically be subclasses of those. Note that you can choose an appropriate base class: If you don't have a form, you don't need a FormController. This is a major difference to Struts.

You can take any object as command or form object: There's no need to implement an interface or derive from a base class. Spring's data binding is highly flexible, e.g. it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. So you don't need to duplicate your business objects' properties as Strings in your form objects, just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it's often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes like Action and ActionForm - for every type of action.

Compared to WebWork, Spring has more differentiated object roles: It supports the notion of a Controller, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data. Instead, a WebWork Action combines all those roles into one single object. WebWork does allow you to use existing business objects as part of your form, but just by making them bean properties of the respective Action class. Finally, the same Action instance that handles the request gets used for evaluation and form population in the view. Thus, reference data needs to be modelled as bean properties of the Action too. These are arguably too many roles in one object.

Regarding views: Spring's view resolution is extremely flexible. A Controller implementation can even write a view directly to the response, returning null as ModelAndView. In the normal case, a ModelAndView instance consists of a view name and a model Map, containing bean names and corresponding objects (like a command or form, reference data, etc). View name resolution is highly configurable, either via bean names, via a properties file, or via your own ViewResolver implementation. The abstract model Map allows for complete abstraction of the view technology, without any hassle: Be it JSP, Velocity, or anything else - every renderer can be integrated directly. The model Map simply gets transformed into an appropriate format, like JSP request attributes or a Velocity template model.

11.1.1. Pluggability of MVC implementation

Many teams will try to leverage their investments in terms of know-how and tools, both for existing projects and for new ones. Concretely, there are not only a large number of books and tools for Struts but also a lot of developers that have experience with it. Thus, if you can live with Struts' architectural flaws, it can still be a viable choice for the web layer. The same applies to WebWork and other web frameworks.

If you don't want to use Spring's web MVC but intend to leverage other solutions that Spring offers, you can integrate the web framework of your choice with Spring easily. Simply start up a Spring root application context via its ContextLoaderListener, and access it via its ServletContext attribute (or Spring's respective helper method) from within a Struts or WebWork action. Note that there aren't any "plugins" involved, therefore no dedicated integration: From the view of the web layer, you'll simply use Spring as a library, with the root application context instance as entry point.

All your registered beans and all of Spring's services can be at your fingertips even without Spring's web MVC. Spring doesn't compete with Struts or WebWork in this usage, it just addresses the many areas that the pure web frameworks don't, from bean configuration to data access and transaction handling. So you are able to enrich your application with a Spring middle tier and/or data access tier, even if you just want to use e.g. the transaction abstraction with JDBC or Hibernate.

11.1.2. Features of Spring MVC

If just focussing on the web support, some of the Spring's unique features are:

- Clear separation of roles: controller vs validator vs command object vs form object vs model object, DispatcherServlet vs handler mapping vs view resolver, etc.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy in-between referencing via an application context, e.g. from web controllers to business objects and validators.
- Adaptability, non-intrusiveness: Use whatever Controller subclass you need (plain, command, form, wizard, multi action, or a custom one) for a given scenario instead of deriving from Action/ActionForm for everything.
- Reusable business code, no need for duplication: You can use existing business objects as command or form objects instead of mirroring them in special ActionForm subclasses.
- Customizable binding and validation: type mismatches as application-level validation errors that keep the offending value, localized date and number binding, etc instead of String-only form objects with manual parsing and conversion to business objects.
- Customizable handler mapping, customizable view resolution: flexible model transfer via name/value Map, handler mapping and view resolution strategies from simple to sophisticated instead of one single way.
- Customizable locale and theme resolution, support for JSPs with and without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, etc.
- Simple but powerful tag library that avoids HTML generation at any cost, allowing for maximum flexibility in terms of markup code.

11.2. General overview

The general overview of the Spring web framework

Chapter 12. Integrating third-party software

12.1. Introduction

Spring is an application framework for all layers: It offers a bean configuration foundation, AOP support, a JDBC abstraction framework, abstract transaction support, etc. It is a very non-intrusive effort: Your application classes do not need to depend on any Spring classes if not necessary, and you can reuse every part on its own if you like to. From its very design, the framework encourages clean separation of tiers, most importantly web tier and business logic: e.g. the validation framework does not depend on web controllers. Major goals are reusability and testability: Unnecessary container or framework dependencies can be considered avoidable evils.

Spring is potentially a one-stop shop, addressing most infrastructure concerns of typical applications. It also goes places other frameworks don't. However, if you like to use other technologies for certain layers in your application (e.g. the web layer or the persistence layer), Spring allows you to replace whatever solution it has for that specific layer, with any other solution. Spring provides a couple of pre-integrated technologies. These are described in this chapter.

12.2. Integrating Velocity

Velocity is a view technology developed the Jakarta Project. More information about Velocity can be found at http://jakarta.apache.org/velocity. This chapter describes how to integrate the Velocity view technology for use with Spring.

12.2.1. Dependencies

There is one dependency that your web application will need to satisfy before working with Velocity, namely that velocity-1.x.x.jar needs to be available. Typically this is included in the WEB-INF/lib folder where it is guaranteed to be found by a J2EE server and added to the classpath for your application. It is of course assumed that you already have the spring-full.jar (or equivalent) in your WEB-INF/lib folder too! The latest stable velocity jar is normally supplied as part of the Spring framework and can be copied from there.

12.2.2. Dispatcher Servlet Context

The configuration file for your Spring dispatcher servlet (usually WEB-INF/[servletname]-servlet.xml) should already contain a bean definition for the view resolver. We'll also add a bean here to configure the Velocity environment. I've chosen to call my dispatcher 'frontcontroller' so my config file names reflect this.

The following code examples show the various configuration files with appropriate comments.

```
<!-- ========================================================-->
<!-- View resolver. Required by web framework.               -->
<!-- ========================================================-->
<!--
        View resolvers can be configured with ResourceBundles or XML files.  If you need
        different view resolving based on Locale, you have to use the resource bundle resolver,
        otherwise it makes no difference.  I simply prefer to keep the Spring configs and
        contexts in XML.  See Spring documentation for more info.
-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
        <property name="cache"><value>true</value></property>
        <property name="location"><value>/WEB-INF/frontcontroller-views.xml</value></property>
</bean>

<!-- ===========================================================-->
<!-- Velocity configurer.                                       -->
<!-- ===========================================================-->
<!--
        The next bean sets up the Velocity environment for us based on a properties file, the
        location of which is supplied here and set on the bean in the normal way.  My example shows
        that the bean will expect to find our velocity.properties file in the root of the
        WEB-INF folder.  In fact, since this is the default location, it's not necessary for me
        to supply it here.  Another possibility would be to specify all of the velocity
        properties here in a property set called "velocityProperties" and dispense with the
        actual velocity.properties file altogether.
-->
<bean
        id="velocityConfig"
        class="org.springframework.web.servlet.view.velocity.VelocityConfigurer"
        singleton="true">
        <property name="configLocation"><value>/WEB-INF/velocity.properties</value></property>
</bean>
```

12.2.3. Velocity.properties

This file contains the values that are passed to the Velocity runtime in order to configure velocity itself. Only a few properties are actually required, but many more optional properties are available - see the Velocity docs for more information. Here, I'm just demonstrating the bare minimum to get Velocity up and running in your Spring MVC application.

The main property values concern the location of the Velocity templates themselves. Velocity templates can be loaded from the classpath or the file system and there are pros and cons for both. Loading from the classpath is entirely portable and will work on all target servers, but you may find that the templates clutter your java packages (unless you create a new source tree for them). A further downside of classpath storage is that during development, changing anything in the source tree often causes a refresh of the resource in the WEB-INF/classes tree and this in turn may cause your development server to restart the application (hot-deploying of code). This can be irritating. Once most of the development is complete though, you could store the templates in a jar file which would make them available to the application if this were placed in WEB-INF/lib.

This example stores velocity templates on the file system somewhere under WEB-INF so that they are not directly available to the client browsers, but don't cause an application restart in development every time I change one. The downside is that the target server may not be able to resolve the path to these files correctly, particularly if the target server doesn't explode WAR modules on the file system. The file method works fine for Tomcat 4.1.x, WebSphere 4.x and WebSphere 5.x. Your mileage may vary.

```
#
# velocity.properties - example configuration
#


# uncomment the next two lines to load templates from the
# classpath (WEB-INF/classes)
#resource.loader=class
#class.resource.loader.class=org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader

# comment the next two lines to stop loading templates from the
# file system
resource.loader=file
file.resource.loader.class=org.apache.velocity.runtime.resource.loader.FileResourceLoader
```

```
# additional config for file system loader only.. tell Velocity where the root
# directory is for template loading.  You can define multiple root directories
# if you wish, I just use the one here.  See the text below for a note about
# the ${webapp.root}
file.resource.loader.path=${webapp.root}/WEB-INF/velocity


# caching should be 'true' in production systems, 'false' is a development
# setting only.  Change to 'class.resource.loader.cache=false' for classpath
# loading
file.resource.loader.cache=false

# override default logging to direct velocity messages
# to our application log for example.  Assumes you have
# defined a log4j.properties file
runtime.log.logsystem.log4j.category=com.mycompany.myapplication
```

The file resource loader configuration above uses a marker to denote the root of the web application on the file system ${webapp.root}. This marker will be translated into the actual OS-specific path by the Spring code prior to supplying the properties to Velocity. This is what makes the file resource loader non-portable in some servers. The actual name of the marker itself can be changed if you consider it important by defining a different "appRootMarker" for VelocityConfigurer. See the Spring documentation for details on how to do this.

12.2.4. View configuration

The last step in configuration is to define some views that will be rendered with velocity templates. Views are always defined in a consistent manner in Spring context files. As noted earlier, this example uses an XML file to define view beans, but a properties file (ResourceBundle) can also be used. The name of the view definition file was defined earlier in our ViewResolver bean - part of the WEB-INF/frontcontroller-servlet.xml file.

```

<!--
  Views can be hierarchical, here's an example of a parent view that
  simply defines the class to use and sets a default template which
  will normally be overridden in child definitions.
-->
<bean id="parentVelocityView" class="org.springframework.web.servlet.view.velocity.VelocityView">
        <property name="templateName"><value>mainTemplate.vm</value></property>
</bean>

<!--
  - The main view for the home page.  Since we don't set a template name, the value
  from the parent is used.
-->
<bean id="welcomeView" parent="parentVelocityView">
        <property name="attributes">
                <props>
                        <prop key="title">My Velocity Home Page</prop>
                </props>
        </property>
</bean>

<!--
  - Another view - this one defines a different velocity template.
-->
<bean id="secondaryView" parent="parentVelocityView">
        <property name="templateName"><value>secondaryTemplate.vm</value></property>
        <property name="attributes">
                        <props>
                                <prop key="title">My Velocity Secondary Page</prop>
                        </props>
        </property>
</bean>
```

12.2.5. Create templates and test

Finally, you simply need to create the actual velocity templates. We have defined views that reference two templates, mainTemplate.vm and secondaryTemplate.vm Both of these files will live in WEB-INF/velocity/ as noted in the velocity.properties file above. If you chose a classpath loader in velocity.properties, these files would live in the default package (WEB-INF/classes), or in a jar file under WEB-INF/lib. Here's what our 'secondaryView' might look like (simplified HTML)

```
## $title is set in the view definition file for this view.
<html>
        <head><title>$title</title></head>
        <body>
                <h1>This is $title!!</h1>
        </body>
</html>
```

Now, when your controllers return a ModelAndView with the "secondaryView" set as the view to render, Velocity should kick in with the above page. Summary To summarize, this is the tree structure of files discussed in the example above. Only a partial tree is shown, some required directories are not highlighted here. Incorrect file locations are probably the major reason for velocity views not working, with incorrect properties in the view definitions a close second.

```
ProjectRoot
        |
        +- WebContent
                |
                +- WEB-INF
                          |
                          +- lib
                          |    |
                          |    +- velocity-1.3.1.jar
                          |    +- spring-full-1.0M1.jar
                          |
                          +- velocity
                          |    |
                          |    +- mainTemplate.vm
                          |    +- secondaryTemplate.vm
                          |
                          +- frontcontroller-servlet.xml
                          +- frontcontroller-views.xml
                          +- velocity.properties
```