Spring Framework Reference Documentation
WORK IN PROGRESS!!!

Version: 1.0

Table of Contents

Chapter 1. Introduction

1.1. Introduction

In short, Spring is a ligthweight container based upon Inversion of Control principles. Inversion of Control is a design pattern for component based architectures that turns over the responsibilities of resolving of dependencies to the container instead of letting the components resolve those themselves. The lightweight aspect of the container shows itself when implementing components for use within Spring. Components developed for Spring will not require any external libraries. Furthermore, the container is lightweight in that it does not have some of the major drawbacks of for instance an EJB container; i.e. long startup times, complex testing, deployment and configuration overhead, etcetera.

This chapter first of all describes the overall design of the lightweight container. Furthermore, it provides a quick overview of some of the features Spring has besides the IoC implementation. In short these are:

- *built-in AOP support* to for instance facilitate declarative transaction management outside an EJB container
- *framework for data access*, whether using JDBC or an O/R mapping product such as Hibernate
- *an MVC web framework* completely integrated with the rest of the Spring framework, providing a clean, non-intrusive way for doing MVC implementation, without tying yourself into one specific view
- *support for sending email* using JavaMail or any other mailing system
- *source level medeta support* to model enterprise services using for example AOP
- *JNDI abstraction layer* to facilitate for example transparent switching between remote and local services

Also, this chapter gives you some guidelines on when to choose Spring for a specific project and provides an overview of some of the advantages (and also disadvantages) of Spring and IoC in general.

1.2. About this reference

Teh Spring reference documentation is (to be) an extensive guide to Spring and should offer full insight into all the Spring features. Since the Spring Framework is quite big, this document is also quite big. This section provides you with some information about how to read this and what sections are important for certain scenarios.

Chapter 2. Background articles

2.1. Inversion of Control

This is supposed to be a background article about Inversion of Control in order to give users a better feeling of why they actually should use IoC in general and Spring specifically.

2.2. Model-View-Controller pattern (MVC)

This is supposed to be a background article about Model View Controller.

Chapter 3. The Bean Package

3.1. Introduction

TODO: My idea is to generally describe the beans package for people to get a basic understanding of what it does. However, this is quite difficult as you might see... I have to take a further look at it... More info on IoC needs to be added and also some docs on the type 2 type 3 stuff...

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.

There are three important concepts in the beans package. First of all there's the `BeanWrapper` which offers functionality for setting and getting properties of JavaBeans. Secondly, the concept of `BeanFactories` is important. A BeanFactory is a generic factory, capable of instantiating objects and manage relationships between different objects. Several different types of beans can be managed and there's support for initialization and other lifecycle methods. The BeanFactory basically instantiates `BeanDefinitions`. Bean definitions are - as the name might incur - the definitions of your beans. A BeanDefinition defines the class of the bean, the mode it will be instantiated in, but also defines what collaborators it needs at runtime. Each of the three concepts (*BeanFactory*, *BeanWrapper* and *BeanDefinition*) will be discussed in much more detail below.

3.2. Bean manipulation and the `BeanWrapper`

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/products/javabeans [http://java.sun.com/products/javabeans/]).

One quite important concept of 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. Then, the BeanWrapper support the ability to add standard JavaBeans `PropertyChangeListeners` and `VetoableChangeListeners`, without the need for supporting code in the target class. Last but not least, the BeanWrapper provides support for the setting of indexed properties. The BeanWrapper usually isn't used by application code directly, but by the `DataBinder` and the `BeanFactory`.

The way the BeanWrapper works is partly indicated by its name: *it wraps a bean* to perform action on that bean, like setting and retrieving properties.

3.2.1. Setting and getting basic and nested properties

Setting and getting properties is done using the `setPropertyValue(s) getPropertyValue(s)` methods that both come with a couple of overloaded variants. They're all described in more detail in the JavaDoc Spring comes with. What's important to know that there's a couple of conventions for indicating properties of an object. A couple of examples:

Table 3.1. Examples of properties

Expression	Explanation
`name`	Indicates the property `name` corresponding to the methods `getName()` or `isName()` and `setName()`
`account.name`	Indicates the nested property `name` of the property `account` corresponding e.g. to the methods `getAccount().setName()` or `getAccount().getName()`
`account[2]`	Indicates the *third* element of the indexed property `account`. Indexed properties can be of type `array`, `list` or other *natural ordered* collection

Below you'll find some examples of working with the BeanWrapper to get and set properties.

Note: this part is not important to you if you're not planning to work with the BeanWrapper directly. If you're just using the `DataBinder` and the `BeanFactory` and their out-of-the-box implementation, don't mind reading this and go on with reading about `PropertyEditors`.

Consider the following two classes:

```
public class Company {
        private String name;
        private Employee managingDirector;

        public String getName() {
                return this.name;
        }
        public void setName(String name) {
                this.name = name;
        }
        public Employee getManagingDirector() {
                return this.managingDirector;
        }
        public void setManagingDirector(Employee managingDirector) {
                this.managingDirector = managingDirector;
        }
}
```

```
public class Employee {
        private float salary;

        public float getSalary() {
                return salary;
        }
        public void setSalary(float salary) {
                this.salary = salary;
        }
}
```

The following code snippets show some examples of how to retrieve and manipulate some of the properties of instantiated: `Companies` and `Employees`

```
Company c = new Company();
BeanWrapper bwComp = BeanWrapperImpl(c);
// setting the company name...
bwComp.setPropertyValue("name", "Some Company Inc.");
// ... can also be done like this:
PropertyValue v = new PropertyValue("name", "Some Company Inc.");
```

```
bwComp.setPropertyValue(v);

// ok, let's create the director and tie it to the company:
Employee jim = new Employee();
BeanWrapper bwJim = BeanWrapperImpl(jim);
bwJim.setPropertyValue("name", "Jim Stravinsky");
bwComp.setPropertyValue("managingDirector", jim);

// retrieving the salary of the managingDirector through the company
Float salary = (Float)bwComp.getPropertyValue("managingDirector.salary");
```

3.2.2. Converting properties using `PropertyEditors`

Sometimes it might be handy to be able to represent properties in a different way than the object itself. For example, a date can be represented in a human readable way, while we're still able to convert the human readable form back to the original date (or even better: convert any date entered in a human readable form, back to Date objects). This behavior can be achieved by *registering custom editors*, of type `java.beans.PropertyEditor`. Registering custom editors to a BeanWrapper gives it the knowledge of how to convert properties to the desired type. Read more about PropertyEditors in the JavaDoc of the `java.beans` package provided by Sun.

An example of working with a PropertyEditor convertin `java.util.Date` objects to a form humans understand:

```
/** Details in this class are excluded for brevity */
public class Person {
        public void setBirthDay(Date d);
        public Date getBirthDay();
}

/** and some other method */
public void doIt() {
        SimpleDateFormat df = new SimpleDateFormat("dd-MM-yyyy");
        // CustomDateEditor located in org.springframework.beans.propertyeditors
        // true indicated that null values are NOT allowed
        CustomDateEditor editor = new CustomDateEditor(df, false);

        Person p = new Person();
        BeanWrapper bw = new BeanWrapper(p);
        bw.registerCustomEditor(editor);
        // this will convert the String to the desired object type: java.util.Date!
        bw.setPropertyValue("birthDay", "22-12-1966");
}
```

The notion of `PropertyEditors` is quite important to for instance the MVC framework, but also other parts of the framework, so in the rest of this document, sometimes references to this part occur. For more information on how to write custom editors yourself, refer to the JavaBeans specification (http://java.sun.com/products/javabeans).

3.2.3. Other features worth mentioning

Besides the features you've seen in the previous sections there a couple of features that might be intereseted to you, though not worth an entire section.

- *determining readability and writability*: using the `isReadable()` and `isWritable()` methods, you can determine whether or not a property is readable or writable
- *retrieving propertydescriptors*: using `getPropertyDescriptor(String)` and `getPropertyDescriptors()` you can retrieve objects of type `java.beans.PropertyDescriptor`, that might come in handy sometimes

3.3. The BeanFactory

The `org.springframework.beans.factory` package and its subpackages provide the basis for Spring's IoC[1] container. Spring's BeanFactory supports IoC type 2 and 3. More information about this can be found further along. The BeanFactory provides a way of obtaining beans by name from a central configuration repository, removing the need individual Java objects to read configuration data from for instance properties files. Configuration of the object is the responsibility of the BeanFactory which makes instances available when needed. Two things are important while considering the BeanFactory. First of all the BeanFactory implementations themselves and how to retrieve beans using the BeanFactory interface. Secondly, the way BeanFactories know about how to instantiate objects and what to do with them before returning them for use. The latter is realized using a concept of bean definitions.

3.3.1. Bean definitions

Bean definitions are the specifications of your beans. Beans are classes, providing functionality, but how the BeanFactory is going to manage your beans and how they are configured, is stated in a bean definition. The following is what actually models the bean definition in order to be able to instantiates beans:

- The beanclass, which is the actual implementation of the bean related to the bean definition
- Bean behavioral configuration elements, which state how the bean should behave in the container (i.e. prototype or singleton, autowiring mode, dependency checking mode, initialization and destruction methods)
- Properties being configuration data for the bean. You could think of the number of connections to use in a bean that manages a connection pool (either specified as properties or as constructor arguments)
- Other beans your bean needs to do its work, i.e. *collaborators* (also specified as properties or as constructor arguments)

The concepts listed above, directly translate to a set of elements the bean definition consists of. These elements are listed below, along with a reference to further documentation about each of them.

Table 3.2. Bean definition explanation

Feature	More info
class	Section 3.3.2, “The bean class”
singleton or prototype	Section 3.3.3, “To singleton or not to singleton”
bean properties	Section 3.3.4, “Setting beans properties and collaborators”
constructor arguments	Section 3.3.4, “Setting beans properties and collaborators”
autowiring mode	Section 3.3.5, “Autowiring collaborators”
dependency checking mode	Section 3.3.6, “Checking for dependencies”
initialization method	Section 3.3.7, “Bean Nature and Lifecycle Features”
destruction method	Section 3.3.7, “Bean Nature and Lifecycle Features”

All the features described above will be configurable for each of your beans using one of the out-of-the-box BeanFactory implementation (like `XmlBeanFactory`). More information about each of the features will be found below.

[1] Inversion of Control, see Section 2.1, “Inversion of Control” for more information

3.3.2. The bean class

Of course you need to specify the actual class of your bean, that should be obvious. There's absolutely no special requirements to your beanclass, it does not have to implement a special interface to make it Spring compatible. Just specifying the bean class should be enough. However, dependending on what type of IoC you're going to use for that specific bean, you should have a default constructor.

The BeanFactory cannot only manage beans, but is able to manage virtually *any* class you want it to manage. Most people using Spring prefer to have actual beans (having just a default constructor and appropriate setters and getters modelled after the properties) in the BeanFactory, but it it's also possible to have more exotic non-bean-style classes in your BeanFactory. If for example you're having a legacy connectionpool that absolutely does not adhere to the bean specification, no worries, Spring can manage it as well.

3.3.3. 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.

Beans exist in singleton mode by default, unless you specify otherwise. Keep in mind that by changing the type to prototype, each request for a bean will result in a newly created bean and this might not be what you actually want. So only change the mode to prototype when absolutely necessary.

3.3.4. Setting beans properties and collaborators

The basic principle of IoC (Inversion of Control) is often referred to as the *Hollywood Principle* (“don't call us, we'll call you!”). The idea is that beans don't specify themselves who they're collaborating with and what additional properties they need. Also the client(s) that use the beans don't specify it. Instead (in case of Spring IoC), the BeanFactory takes care of resolving collaborators and giving them to the beans. This is done using the bean definitions discussed earlier. In the BeanDefinition structure, collaborators and properties are specified using `PropertyValue` objects, which are used when insantiating a bean to resolve references.

It's very important to understand the concept of IoC beacuse this is one of the basics of Spring and will make your applications much more elegant, configurable and maintainable. Configure each and every collaborator and property using the BeanDefinitions (for more information about the implementation have a look at one of the next sections)!

The resolving of the dependencies is a little too complex to go get into it in depth here, the basic procedure is as follows:

1. Checking what the type of the property is (this can be a primitive-like type - like int or String - or a collection - i.e. Map or List - or a collaborator). Collaborators are other beans the BeanFactory must capable of resolving
2. In case of the first two options (primitive or collection), Spring constructs the collection and adds it to the bean definition as a PropertyValue
3. In case of a collaborator, Sprign constructs a `RuntimeBeanReference` that is used later on when the bean is actually instantiated. Spring then actually resolves the instance of the collaborator and sets it on the bean

The setting of properties and collaborators can be done in two ways, which correspond to two of the different IoC types[2], namely type 2 and type 3. The type 2 support, will set the dependencies on a bean using setter methods, IoC type 3 support will set the dependencies using constructor arguments. It's best to explain things using a concrete implementation of a BeanFactory. We will use the `XmlBeanFactory`[3] here, which - as the name implies - stores bean definition in XML.

First, an example of using the BeanFactory for IoC type 2 (using setters). Below, there's a small part of an XML file specifying bean definition. Also, you can find the actual bean itself, having the appropriate setters declared.

```
<bean id="exampleBean" class="examples.ExampleBean">
        <property name="beanOne"><ref bean="anotherExampleBean"/></property>
        <property name="beanTwo"><ref bean="yetAnotherBean"/></property>
        <property name="integerProperty">1</property>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

        private AnotherBean beanOne;
        private YetAnotherBean beanTwo;
        private int i;

        public void setBeanOne(AnotherBean beanOne) {
                this.beanOne = beanOne;
        }

        public void setBeanTwo(YetAnotherBean beanTwo) {
                this.beanTwo = beanTwo;
        }

        public void setIntegerProperty(int i) {
                this.i = i;
        }
}
```

As you can, setters have been declared to match against the properties specified in the XML file. (The properties from the XML file, directly relate to the `PropertyValues` object from the `RootBeanDefinition`)

Then, an example of using the BeanFactory for IoC type 3 (using constructors). Below you can find a snippet from an XML configuration file that specifies constructor arguments and the actual bean, specifying the constructor

```
<bean id="exampleBean" class="examples.ExampleBean">
        <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>
        <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>
        <constructor-arg>1</constructor-arg>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```

```
public class ExampleBean {

        private AnotherBean beanOne;
        private YetAnotherBean beanTwo;
```

[2] For more information about IoC types, see Section 2.1, “Inversion of Control”

[3] See Section 3.4, “BeanFactory implementations” for more information about BeanFactory implementations

```
        private int i;

        public ExampleBean(AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {
                this.beanOne = anotherBean;
                this.beanTwo = yetAnotherBean;
                this.i = i;
        }
}
```

As you can see, the constructor arguments specified in the bean definition will be used to pass in as arguments to the constructor of the `ExampleBean`.

3.3.5. Autowiring collaborators

Spring has autowire capabilities, which means it's possible to automatically let Spring resolve collaborators (other beans) for you bean by inspecting the BeanFactory. The autowiring functionality has four modes. Autowiring is specified *per* bean and can thus be enabled for a couple of beans, while other beans won't be autowired. When using autowiring, there might be no need for specifying properties or constructor arguments[4]

Table 3.3. Autowiring modes

Mode	Explanation
no	No autowiring at all. This is the default value and it's discouraged to change this for large applications, since specifying your collaborators yourself gives you a feeling of what you're actually doing and is a great way of somewhat documenting the structure of your system
byName	This option will inspect the BeanFactory and look for a bean named exactly the same as the property which needs to be autowired. So in case you have a collaborator on a BeanDefinition Cat which is called dog (so you have a setDog(Dog) method), Spring will look for a BeanDefinition named `dog` and use this as the collaborator
byType	This option can be found in some other IoC containers as well and gives you the ability to resolve collaborators by type instead of by name. Suppose you have a BeanDefinition with a collaborator typed `DataSource`, Spring will search the entire bean factory for a bean definition of type DataSource and use it as the collaborator. *If 0 (zero) or more than 1 (one) bean definitions of the desired exist in the BeanFactory, a failure will be reported and you won't be able to use autowiring for that specific bean*

Note: like already mentioned, for larger application it's discourage to use autowiring beacuse it removes the transparency and the structure from your collaborating classes.

3.3.6. Checking for dependencies

Spring also offers the capability for checking required dependencies of your beans. This feature might come in handy when certain properties really need to be set and when you can't provide default values (which is an often used approach). Dependency checking can be done in three different ways. Dependency checking can also be enabled and disabled per bean, just as the autowiring functionality. The default is to *not* check dependencies.

Table 3.4. Dependency checking modes

[4] See Section 3.3.4, “Setting beans properties and collaborators”

Mode	Explanation
simple	Dependency checking is done for primitive types and collections (this means everything except collaborators)
object	Dependency checking is done for collaborators
all	Dependecny checking is done for both collaborators and primitive types and collections

3.3.7. Bean Nature and Lifecycle Features

Spring provides a couple of marker interfaces and some other features that allow customizing the nature and 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.

3.3.7.1. 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

3.3.7.2. 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)

3.3.7.3. 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????

3.3.7.4. 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)

3.3.7.5. 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????

3.3.7.6. 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).

3.3.8. 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 :)

3.3.9. Lifecycle of a bean in the BeanFactory

This section describes the lifecycle of a bean in the BeanFactory, some of the basic characteristics of events happening in the BeanFactory as well as the different types of beans.

3.3.9.1. The basic lifecycle of a bean

The lifecycle of a bean begins with a bean definition, for instance defined in XML or a properties file. The first step is the calling of the default constructor:

1. default constructor

The second step is the initialization process in which you can prepare your bean for use.

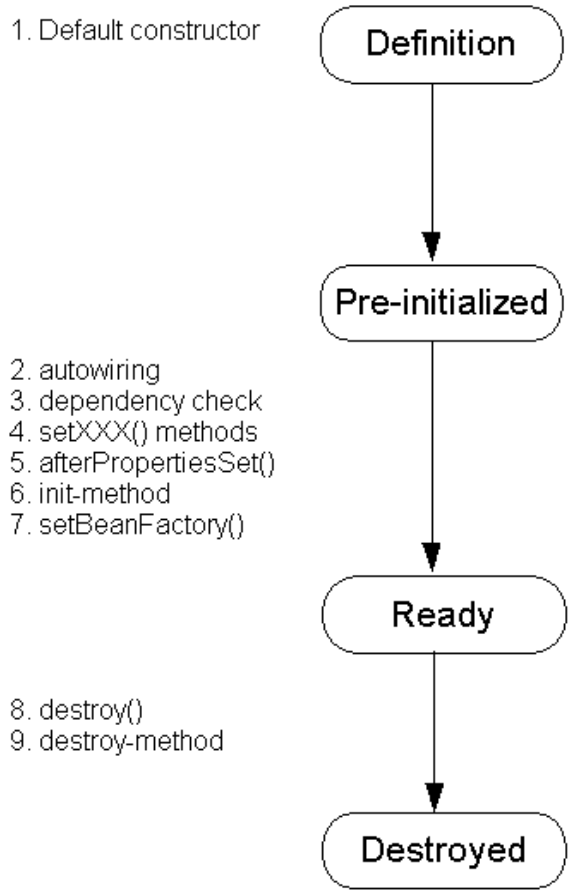
2. the autowiring process in which any possible collaborators that can be automatically resolved, will be set (TODO: reference)
3. checking of dependencies, which means that if dependencies are satisfied (i.e. they are `null`), an `UnsatisfiedDependencyException` will be thrown. (TODO: reference)
4. property setting, which means that all properties defined with the bean definition in for instance an XML file, will be set on the bean
5. `afterPropertiesSet()` is called. This method is specified by the `InitializingBean`, and thus will only be called if your bean implements this interface (TODO: reference)
6. the extra initialization method will be called that might have been specified with the bean definition (TODO: reference)
7. if your bean a `BeanFactoryAware` the `setBeanFactory()` method will now be called enabling the bean to have access to the BeanFactory (TODO: reference)

Right now, your bean is ready for use. Calls to `BeanFactory.getBean()` with the name of the bean, will result in an instance of that bean being returned. Depending on whether or not this bean is a singleton, a shared instance will be returned. More information on consequences for singletons can be found below.

When the `BeanFactory` gets destroyed (for instance when the application server shuts down, the destruction process of the BeanFactory will try to destroy all beans that it still knows. The destruction process *only comprises singletons beans.*

8. If your bean implements `DisposableBean` the BeanFactory will call `destroy()`
9. If you bean definition includes a destroy method declaration, this method will be called as well

A diagram illustrating the lifecycle of a bean:



3.4. BeanFactory implementations

A couple of implementations of the BeanFactory come out-of-the-box. The `XmlBeanFactory` supports a bean definitions to be specified in XML files and the `ListableBeanFactory` supports bean definitions in the form of

properties files. Most people use the XmlBeanFactory. However, implementing your own BeanFactory that supports bean definition in a database should not be to big an issue. Let's first discuss the XmlBeanFactory and the ListableBeanFactory and their features.

Basically the two BeanFactory implementations Spring comes with provide all the features described above, like specifying the lifecycle methods, specifying whether or not to do autowiring, etcetera. The only way they differ is the way the configuration data (the bean definitions) are stored.

3.4.1. Bean definitions specified in XML (`XmlBeanFactory`)

One of the implementations of the BeanFactory is the `XmlBeanFactory` (located in the package `org.springframework.beans.factory.xml`) which offers you the ability to specify bean definition in XML files as the name might have already told you. Spring comes with a DTD to do validation of the XML you're writing to specify beans in order to make things a bit more easy. The DTD is quite well documented so you should find everything you need in there as well. Here we will discuss the format shortly and provide some examples.

The root of a Spring XML bean definition document is a `<beans>` element. The `<beans>` element contains one or more `<bean>` definitions. We normally specify the class and properties of each bean definition. We must also specify the id, which will be the name that we'll use this bean with in our code (see a previous section about clients interacting with the BeanFactory for more information. An initialization method as described earlier as well the destruction method can be specified as attributes of the `<bean>` element. The autowiring functionality as well as the de dependency checking can also be specified using attributes of the same element. Furthermore properties and collaborators can be specified using nested `<property>` elements. In the following example, we use a BasicDataSource from the Jakarta Commons DBCP project. This class (like many other existing classes) can easily be used in a Spring bean factory, as it offers JavaBean-style configuration. The close method that needs to be called on shutdown can be registered via Spring's “destroy-method” attribute, to avoid the need for BasicDataSource to implement any Spring interface (in this case that would be `DisposableBean` mentioned earlier in the section about lifecycle features).

```
<beans>
        <bean id="myDataSource"
                class="org.apache.commons.dbcp.BasicDataSource"
                destroy-method="close">
                <!-- results in a setDriverClassName(String) call -->
                <property name="driverClassName">
                        <value>com.mysql.jdbc.Driver</value>
                </property>
                <property name="url">
                        <value>jdbc:mysql://localhost:3306/mydb</value>
                </property>
                <property name="username">
                        <value>root</value>
                </property>
        </bean>
</beans>
```

Just as an destruction method is specified using the `destroy-method` attribute, we could specify a initialization method using the `init-method` attribute.

To specify properties and collaborators in XML files you have to use nested <property> elements. You have already seen the setting of 'primitive' properties in the example about, the setting of collaborators is done using the nested `<ref>` element.

```
<beans>
...
        <bean id="exampleDataAccessObject"
                class="example.ExampleDataAccessObject">
```

```
                <!-- results in a setDataSource(BasicDataSource) call -->
                <property name="dataSource">
                        <ref bean="myDataSource"/>
                </property>
        </bean>
</beans>
```

As you can see below, we're using the Commons DBCP datasource from the previous example here as a collaborator and we're specifying if using a <ref bean> element. References exist in three types that specify whether or not to search the collaborator in the same XML file or in some other XML file (multiple XML files is covered further along):

- `bean`: tries to find the collaborator in either the same XML file or in some other XML file that has also been specified
- `local`: tries to find the collaborator in the current XML file. This attribute is an `XML IDREF` so it *has* to exist, otherwise validation will fail
- `external`: explicitly states to find the bean in another XML file and does not search in the current XML file

There's a couple of possibilities for specifying more complex properties such as lists, properties object and maps. The following examples show this behavior:

```
<beans>
...
        <bean id="moreComplexObject"
                class="example.ComplexObject">
                <!-- results in a setPeople(java.util.Properties) call -->
                <property name="people">
                        <props>
                                <prop key="HaaryPotter">The magic property</prop>
                                <prop key="JerrySeinfeld">The funny property</prop>
                        </props>
                </property>
                <!-- results in a setSomeList(java.util.List) call -->
                <property name="someList">
                        <list>
                                <value>a list element followed by a reference</value>
                                <ref bean="myDataSource"/>
                        </list>
                </property>
                <!-- results in a setSomeMap(java.util.Map) call -->
                <property name="someMap">
                        <map>
                                <entry key="yup an entry">
                                        <value>just some string</value>
                                </entry>
                                <entry key="yup a ref">
                                        <ref bean="myDataSource</ref>
                                </entry>
                        </map>
                </property>
        </bean>
</beans>
```

Note that the value of a Map entry can also again be a list or another map.

Chapter 4. The Context Package

4.1. Introduction

On top of the beans package, providing basic functionality for managing and manipulating beans, sits the `context` package. The context package acts as a registry of objects needed by your application. The functionality it offers somewhat resembles the *enterprise naming context* J2EE application servers feature, however there are some key differences. For instance, objects bound in a Spring application context (`ApplicationContext`) don't need to extend something like a remote interface or object. Second of all, application contexts can be used throughout the complete range of layers of your applications, from within Swing clients, webapplications, but also in Enterprise JavaBeans.

The basis for the context package is the `org.springframework.context.ApplicationContext` interface, providing functionality for internationalization, eventhandling and beans being aware of the context they're existing in. Also it's possible to create a hierachical structure of contexts, enabling beans to be scoped for and only accessible to a certain part of an application.

4.2. The ApplicationContext managing beans

The ApplicationContext includes all the functionality the bean factory has as well. This means you can define beans using the XmlBeanFactory and look up bean by specifying their name or id. All features the BeanFactory offers are available to you, including the the lifecycle interfaces, the initialization and destruction methods, dependency checking and autowiring. The BeanFactory will not be discussed in this chapter, instead the ApplicationContext specific features will be described here. In the previous chapter, there's more information on the specifics of the `BeanFactory`.

4.3. ApplicationContext basics

The ApplicationContext, provides - as described above

Chapter 5. Validating and data binding

5.1. Introduction

The big question is whether or not validation should be considered *business logic*. There's pros and cons for both answers and Spring offers a design for validation (and data binding) that does not exclude either one of them. Validation should specifically not be tied to the web tier, should be easily localizable and it should be possible to plug in any validator available. Considering the above, Spring has come up with a `Validator` interface that's both basic and usable in every layer of an application.

Data binding is useful for allowing user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-called `DataBinder` to do exactly that. The Validator and the DataBinder make up the `validation` package, which is primarily used in the MVC framework.

5.2. Binding data using the `DataBinder`

The DataBinder builds on top the BeanWrapper[5]

[5] See the beans chapter for more information

Chapter 6. Aspect Oriented Programming with Spring

Chapter 7. Sourcelevel Metadata Support

Chapter 8. Web framework

8.1. 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, locale and theme resolution as well support for upload files. 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 for example `AbstractController`, `AbstractCommandController` and `SimpleFormController`. 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.

8.1.1. 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 appli-

cation 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.

8.1.2. 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.

8.2. The `DispatcherServlet`

Spring's web framework is - like many other web frameworks - designed around a servlet that dispatches requests to controller and offers other functionality facilitating the development of webapplications. Spring's `DispatcherServlet` however, does more than just that. It is completely integrated with the Spring ApplicationContext and makes fully use of that.

Servlets are declared in the `web.xml` of your webapplication, so is the DispatcherServlet. Requests that you want the DispatcherServlet to handle, will have to be mapped, using a url-mapping in the same `web.xml` file.

```
<web-app>
        ...
        <servlet>
                <servlet-name>example</servlet-name>
                <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
                <load-on-startup>1</load-on-startup>
        </servlet>
        <servlet-mapping>
                <servlet-name>example</servlet-name>
                <url-pattern>*.form</url-pattern>
        </servlet-mapping>
</web-app>
```

In the example above, all requests ending with `.form` will be handled by the DispatcherServlet. Then, the DispatcherServlet needs to be configured. As illustrated in Chapter 4, *The Context Package*, ApplicationContexts in Spring can be scoped. In the web framework, each DispatcherServlet has its own `WebApplicationContext`, which contains the DispatcherServlet configuration beans. The default BeanFactory used by the DispatcherServlet is the `XmlBeanFactory` and the DispatcherServlet will on initialization *look for a file named* `[servlet-name]-servlet.xml` in the `WEB-INF` directory of your web application. The default values used by the DispatcherServlet can be modified by using the servlet initialization parameters (see below for more information).

The `WebApplicationContext` is just an ordinary ApplicationContext that has some extra features necessary for webapplications. It differs from a normal ApplicationContext in that it is capable of resolving themes (see ???) and that is knows to which servlet it is associated (by having a link to the `ServletContext`). The WebApplicationContext is bound in the ServletContext, and using `RequestContextUtils` you can always lookup the WebApplicationContext in case you need it.

The Spring DispatcherServlet has a couple of special beans it uses, in order to be able to process requests and render the appropriate views. Those beans are included in the Spring framework and (optionally) have to be configured in the WebApplicationContext, just as any other bean would have to be configured. Each of those beans, is described in more detail below. Right now, we'll just mention them, just to let you know they exist and to enable us to go on talking about the DispatcherServlet. For most of the beans, defaults are provided so you don't have to worry about those.

Table 8.1. Special beans in the WebApplicationContext

Expression	Explanation
handler mapping(s)	(Section 8.4, “Handler mappings”) a list of pre- and postprocessors and controllers that will be executed if they match certain criteria (for instance a matching URL specified with the controller)
controller(s)	(???) the beans providing the actual functionality (or at least, access to the functionality) as part of the MVC triad
view resolver	(Section 8.5, “Views and resolving them”) capable of resolving view names and needed by the DispatcherServlet to resolves those views with
locale resolver	(Section 8.6, “Using locales”) capable of resolves the locale a client is using, in order to be able to offer internationalized views
theme resolver	(Section 8.7, “Using themes”) capable of resolving themes your webapplication can use e.g. to offer personalized layouts
multipart resolver	(Section 8.8, “Spring's multipart (fileupload) support”) offers functionality to process file uploads from HTML forms

When a DispatcherServlet is setup for use and a request comes in for that specific DispatcherServlet it starts processing it. The list below describes the complete process a request goes through if a DispatcherServlet is supposed to handle it:

1. The WebApplicationContext is searched for and bound in the request as an attribute in order for controller and other elements in the chain of process to use it. It is bound by default under the key `DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE`
2. The locale resolver is bound to the request to let elements in the chain resolve the locale to use when processing the request (rendering the view, preparing data, etcetera). If you don't use the resolver, it won't af-

fect anything, so if you don't need locale resolving, just don't bother
3. The theme resolver is bound to the request to let e.g. views determine which theme to use (if you don't needs themes, don't bother, the resolver is just bound and does not affect anything if you don't use it)
4. If a multipart resolver is specified, the request is inspected for multiparts and if so, it is wrapped in a `MultipartHttpServletRequest` for further processing by other elements in the chain (more information about multipart handling is provided below)
5. An appropriate handler is searched for. If a handler is found, it execution chain associated to the handler (preprocessors, postprocessors, controllers) will be executed in order to prepare a model
6. If a model is returned, the view is rendered, using the view resolver that has been configured with the WebApplicationContext. If no model was returned (which could be the result of a pre- or postprocessor intercepting the request because of for instance security reasons), no view is rendered as well, since the request could already have been fulfilled

The Spring DispatcherServlet also has support for returning the *last-modification-date*, as specified by the Servlet API. The process of determining the last modification date for a specific request, is simple. The DispatcherServlet will first of all lookup an appropriate handler mapping and test is the handler that matched *implements the interface* `LastModified` and if so, the value the of `long getLastModified(request)` is returned to the client.

TODO: initialization parameters of the servlet

8.3. Controllers

The notion of controller is part of the MVC design pattern. Controllers define application behavior, or at least provide users with access to the application behavior. Controllers interpret user input and transform the user input into a sensible model which will be represented to the user by the view. Spring has implemented the notion of a controller in a very abstract way enabling a wide variety of different kinds of controllers to be created. Spring contains formcontroller, commandcontroller, controllers that execute wizard-style logic and more.

Spring's basis for the controller architecture is the `org.springframework.mvc.Controller` interface, which is listed below.

```
public interface Controller {

        /**
         * Process the request and return a ModelAndView object which the DispatcherServlet
         * will render.
         */
        ModelAndView handleRequest(
                HttpServletRequest request,
                HttpServletResponse response)
        throws Exception;
}
```

As you can see, the Controller interface just states one single method that should be capable of handling a request and return an appropriate model and view. Those three concepts are the basis for the Spring MVC implemente; *ModelAndView* and *Controller*. While the Controller interface is quite abstract, Spring offers a lot of controllers that already contain a lot of functionality you might need. The controller interface just define the most commons functionality offered by every controller: the functionality of handling a request and returning a model and a view.

8.3.1. Using the `AbstractController` and `WebContentGenerator`

Of course, jsut a controller interface isn't enough. To provide a basic infrastructure, all Spring's Controller in-

herit from AbstractController, a class offering caching support and for instance the setting of the mimetype.

Table 8.2. Features offered by the `AbstractController`

Feature	Explanation
`supportedMethods`	indicates what methods this controller should accept. Usually this is set to both `GET` and `POST`, but you can modify this to reflect the method you want to support. If a request is received with a method that is not supported by the controller, the client will be informed of this (using a `ServletException`))
`requiresSession`	indicates whether or not this controller requires a session to do its work. This feature is offered to all controllers. If a session is not present when such a controller receives a request, the user is informed using a `ServletException`
`synchronizeSession`	use this if you want handling by this controller to be synchronized on the user's session. To be more specific, extending controller will override the `handleRequestInternal` method, which will be synchronized if you specify this variable
`cacheSeconds`	when you want a controller to generate caching directive in the HTTP response, specify a positive integer here. By default it is set to *-1* so no caching directives will be included
`useExpiresHeader`	tweaking of your controllers specifying the HTTP 1.0 compatible *"Expires"* header. By default it's set to true, so you won't have to touch it
`useCacheHeader`	tweaking of your controllers specifying the HTTP 1.1 compatible *"Cache-Control"* header. By default this is set to true so you won't really have to touch it

the last two properties are actually part of the `WebContentGenerator` which is the superclass of `AbstractController` but to keeps things clear...

When using the AbstractController as a baseclass for your controllers (which is *not* recommended since there are a lot of other controller that might already do the job for your) you only have to override the `handleRequestInternal(HttpServletRequest, HttpServletResponse)`-method and implement your logic code and return a `ModelAndView` object there. A short example consisting of a class and a declaration in the webapplicationcontext.

```
package samples;

public class SampleController extends AbstractController {

        public ModelAndView handleRequestInternal(
                HttpServletRequest request,
                HttpServletResponse response)
        throws Exception {

                ModelAndView mav = new ModelAndView("index", null);
        }
}
```

```
<bean id="sampleController" class="samples.SampleController">
        <property name="cacheSeconds"><value>120</value</property>
</bean>
```

The class above and the declaration in the webapplicationcontext is all you need to do besides setting up a handler mapping (see Section 8.4, “Handler mappings”) to get this very simple controller working. This controller will generates caching directives telling the client to cache things for 2 minutes before rechecking. This controller furthermore returns an hard-coded view (hmm, not so nice), named index (see Section 8.5, “Views and resolving them” for more information about views).

8.3.2. Other simple controller

Besides the AbstractController - which you could of course extend, although a more concrete controller might offer you more functionality - there are a couple of other simple controllers that might easy the pain of developing simple MVC applications. The `ParameterizableViewController` basically is the same as the one in the example above, except for the fact that you can specify its view name that it'll be returning in the webapplicationcontext (ahhh, no need to hard-code the viewname). The viewname you

The `FileNameViewController` inspects the URL and retrieves the filename of the file request (the filename of `http://www.springframework.org/index.html` is `index`) and uses that as a viewname. Nothing more to it.

8.3.3. The `MultiActionController`

Spring offers a multi-action controller with which you aggregate multiple actions into one controller, grouping functionality together. The multi-action controller lives in a separate package - `org.springframework.web.mvc.multiaction` - and is capable of mapping requests to method names and then invoking the right method name. Using the multi-action controller is especially handy when you're having a lot of commons functionality in one controller, but want to have multiple entry points to the controller to tweak behavior for instance.

Table 8.3. Features offered by the `MultiActionController`

Feature	Explanation
`delegate`	there's two usage-scenarios for the MultiActionController. Either you subclass the MultiActionController and specify the methods that will be resolved by the MethodNameResolver on the subclass (in case you don't need this configuration parameter), or you define a delegate oject, on which methods resolved by the Resolver will be invoked. If you choose to enter this scenario, you will have to define the delegate using this configuration parameter as a collaborator
`methodNameResolver`	somehow, the MultiActionController will need to resolve the method it has to invoke, based on the request that came in. You can define a resolver that is capable of doing that using this configuration parameter

Methods defined for a multi-action controller will need to conform to the following signature:

```
// actionName can be replaced by any methodname
ModelAndView actionName(HttpServletRequest, HttpServletResponse);
```

Method overloading is not allowed since it'll confuse the MultiActionController. Furthermore, you can define *exception handlers* capable of handling exception that will be thrown form a method you specify. Exception handler methods need to return a ModelAndView object, just as any other action method and will need to conform to the following signature:

```

```

```
// anyMeaningfulName can be replaced by any methodname
ModelAndView anyMeaningfulName(HttpServletRequest, HttpServletResponse, ExceptionClass);
```

The `ExceptionClass` can be *any* exception, as long as it's a subclass of `java.lang.Exception` or `java.lang.RuntimeException`.

The `MethodNameResolver` is supposed to resolve method names based on the request coming in. There are three resolver to your disposal, but of course you can implement more of them yourself if you want.

- `ParameterMethodNameResolver` - capable of resolving a request parameter and using that as the method name (`http://www.sf.net/index.view?testParam=testIt` will result in a method `testIt(HttpServletRequest, HttpServletResponse)` being called). Use the `paramName` configuration parameter to tweak the parameter that's inspected)
- `InternalPathMethodNameResolver` - retrieves the filename from the path and uses that as the method name (`http://www.sf.net/testing.view` will result in a method `testing(HttpServletRequest, HttpServletResponse)` being called)
- `PropertiesMethodNameResolver` - uses a user-defined properties object with request URLs mapped to methodnames. When the properties contain `/index/welcome.html=doIt` and a request to `/index/welcome.html` comes in, the `doIt(HttpServletRequest, HttpServletResponse)` method is called. This method name resolver works with the `PathMatcher` (see ???) so if the properties contained `/**/welcom?.html` it would also have worked!

A couple of examples. First of all one showing the `ParameterMethodNameResolver` and the delegate property, which will accept requests to urls with the parameter method included and set to `retrieveIndex`:

```
<bean id="paramResolver" class="org....mvc.multiaction.ParameterMethodNameResolver">
        <property name="paramName"><value>method</value></property>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
        <property name="methodNameResolver"><ref bean="paramResolver"/></property>
        <property name="delegate"><ref bean="sampleDelegate"/>
</bean>

<bean id="sampleDelegate" class="samples.SampleDelegate"/>

## together with

public class SampleDelegate {

        public ModelAndView retrieveIndex(
                HttpServletRequest req,
                HttpServletResponse resp) {

                rerurn new ModelAndView("index", "date", new Long(System.currentTimeMillis()));
        }
}
```

When using the delegates shown above, we could also use the `PropertiesMethodNameRseolver` to match a couple of URLs to the method we defined:

```
<bean id="propsResolver" class="org....mvc.multiaction.PropertiesMethodNameResolver">
        <property name="mappings">
                <props>
                        <prop key="/index/welcome.html">retrieveIndex</prop>
                        <prop key="/**/notwelcome.html">retrieveIndex</prop>
                        <prop key="/*/user?.html">retrieveIndex</prop>
                </props>
        </property>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
        <property name="methodNameResolver"><ref bean="propsResolver"/></property>
```

```
        <property name="delegate"><ref bean="sampleDelegate"/>
</bean>
```

8.3.4. CommandControllers

Spring's *CommandControllers* are a fundamental part of the Spring MVC package. Command controllers provide a way to interact with dataobjects and dynamically bind parameters from the `HttpServletRequest` to the dataobject you're specifying. This compares to Struts actionforms, where in Spring, you don't have to implement any interface of superclasses to do databinding. First, let's examine what command controllers available, just to get clear picture of what you can do with them:

- `AbstractCommandController` - a command controller you can use to create your own command controller, capable of binding request parameters to a data object you're specifying. This class does not offer form functionality, it does however, offer validation features and let's you specify in the controller itself what to do with the dataobject that has been filled with the parameters from the request.
- `AbstractFormController` - an abstract controller offering form submission support. Using this controller you can model forms and populate them using a dataobject you're retrieving in the controller. After a user has filled the form, the AbstractFormController binds the fields, validates and hands the object back to you - the controller - to take appropriate action. Supported features are invalid form submission (resubmission), validation, and the right workflow a form always has. What views you tie to your AbstractFormController you decide yourself. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the applicationcontext
- `SimpleFormController` - an even more concrete FormCotnroller that helps you creating a form with corresponding data object even more. The SimpleFormController let's you specify a command object, a viewname for the form, a viewname for page you want to show the user when formsubmission has succeeded, and more
- `WizardFormController` - last but not least, a WizardFormController allows you to model wizard-style manipulation of dataobjects, which is extremely handy when large dataobjects come in

8.4. Handler mappings

8.5. Views and resolving them

8.6. Using locales

8.7. Using themes

8.8. Spring's multipart (fileupload) support

8.8.1. Introduction

Spring has built-in multipart support to handle fileuploads in webapplications. The design for the multipart support is done in such a way that pluggable so-called `MultipartResovlers` can be used. Out of the box, Spring

provides MultipartResolver for use with *Commons FileUpload* and *COS FileUpload*. How uploading files is supported will be described in the rest of this chapter.

As already said, the multipart support is provided through the `MultipartResolver` interface, located in the `org.springframework.web.multipart` package. By default, no multipart handling will be done by Spring. You'll have to enable it yourself by adding a multipartresolver to the webapplication's context. After you've done that, each request will be inspected for a multipart that it might contain. If no such multipart is found, the request will continue as expected. If however, a multipart is found in the request, the MultipartResolver that has been declared in your context will resolve. After that, the multipart attribute in your request will be treated as any other attributes.

8.8.2. Using the `MultipartResolver`

To be able to resolve multiparts from a request, you will have to declare a MultipartResolver. There's two multipart resolver Spring's comes with. First of all the resolver that works with Commons FileUpload (http://jakarta.apache.org/commons/fileupload), and secondly, the resolver that uses the O'Reilly COS package (http://www.servlets.com/cos). Without a MultipartResolver declared in your webapplication context, no detection of multiparts will take place, because some developers might find the need to parse out the multiparts themselves.

The following example shows how to use the `CommonsMultipartResolver`:

```
<bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver">

        <!-- one of the properties available; the maximum file size in bytes -->
        <property name="maximumFileSize">
                <value>100000</value>
        </property>
</bean>
```

But you can also use the `CosMultipartResolver`:

```
<bean id="multipartResolver"
        class="org.springframework.web.multipart.cos.CosMultipartResolver">

        <!-- one of the properties available; the maximum file size in bytes -->
        <property name="maximumFileSize">
                <value>100000</value>
        </property>
</bean>
```

Of course you need to stick the appropriate jars in your classpath if using one the multipartresolver. In case of the CommonsMultipartResolver, you need to use `commons-fileupload.jar`, in case of the CosMultipartResolver, use `cos.jar`.

Now that Spring's set up to handle multipart requests, let's talk about how to actually use it. When the Spring DispatcherServlet detects a Multipart request, it activates the resolver that has been declared in your context and hands over the request. What it basically does is wrap the current `HttpServletRequest` into a `MultipartHttpServletRequest` that has support for multiparts. Using the MultipartHttpServletRequest you can get information about the multiparts contained by this request and actually get the multiparts themselves in your controllers.

8.8.3. Handling a fileupload in a form

After the MultipartResolver has finished doing its jobs, the request will be processed as any other. So in fact, you can create a form, with a form upload field, and let Spring bind the file on your form. Just as with any other property that's not automagically convertible to a String or primitive type, to be able to put binary data in your beans, you have to register a custom editor with the `ServletRequestDatabinder`. There's a couple of editors available for handling file and setting the result on a bean. There's a `StringMultipartEditor` capable of converting files to Strings (using a user-defined character set) and there's a `ByteArrayMultipartEditor` which concerts files to byte-arrays. They function just as for instance the `CustomDateEditor`

So, to be able to upload files using a form in a website, declare the resolver, a url mapping to a controller that will process the bean and the controller itself.

```
<beans>

        ...

        <bean id="multipartResolver"
                class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

        <bean id="urlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
                <property name="mappings">
                        <props>
                                <prop key="/upload.form">fileUploadController</prop>
                        </props>
                </property>
        </bean>

        <bean id="fileUploadController" class="examples.FileUploadController">
                <property name="commandClass"><value>examples.FileUploadBean</value></property>
                <property name="formView"><value>fileuploadform</value></property>
                <property name="successView"><value>confirmation</value></property>
        </bean>

</beans>
```

After that, create the controller and the actual bean holding the file property

```
// snippet from FileUploadController
public class FileUploadController extends SimpleFormController {

        protected ModelAndView onSubmit(
                HttpServletRequest request,
                HttpServletResponse response,
                Object command,
                BindException errors)
                throws ServletException, IOException {

                // cast the bean
                FileUploadBean bean = (FileUploadBean)command;

                // let's see if there's content there
                byte[] file = bean.getFile();
                if (file == null) {
                        // hmm, that's strange, the user did not upload anything
                }

                // well, let's do nothing with the bean for now and return:
                return super.onSubmit(request, response, command, errors);
        }

        protected void initBinder(
                HttpServletRequest request,
                ServletRequestDataBinder binder)
                throws ServletException {
                // to actually be able to convert Multipart instance to byte[]
                // we have to register a custom editor (in this case the
                // ByteArrayMultipartEditor
                binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
                // now Spring knows how to handle multipart object and convert them
```

```
        }

}

// snippet from FileUploadBean
public class FileUploadBean {
        private byte[] file;

        public void setFile(byte[] file) {
                this.file = file;
        }

        public byte[] getFile() {
                return file;
        }
}
```

As you can see, the FileUploadBean has a property typed byte[] that holds the file. The controller register a custom editor to let Spring know how to actually convert the multipart objects the resolver has found, to properties specified by the bean. Right now, nothing is done with the byte[] and the bean itself, but you can do with it whatever you want (save it in a database, mail it to somebody, etcetera).

But we're still not finished. To actually let the user upload something, we have to create a form:

```
<html>
        <head>
                <title>Upload a file please</title>
        </head>
        <body>
                <h1>Please upload a file</h1>
                <form method="post" action="upload.form" enctype="multipart/form-data">
                        <input type="file" name="file"/>
                        <input type="submit"/>
                </form>
        </body>
</html>
```

As you can see, we've created a field named after the property of the bean that holds the byte[]. Furthermore we've added the encoding attribute which is necessary to let the browser know how to encode the multipart fields (dont' forget this!). Right now everything should work.

8.9. Commonly used utilities

8.9.1. A little story about the pathmatcher

bla

Chapter 9. Integrating third-party software

9.1. 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.

9.2. 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.

9.2.1. 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.

9.2.2. 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>
```

9.2.3. 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.

9.2.4. 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>
```

9.2.5. 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
```

Chapter 10. Sending Email