java/j2ee Application Framework

Reference Documentation

Version: 1.0

(Work in progress)

Copyright (c) 2004 - Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Darren Davison, Dmitriy Kopylenko, Thomas Risberg

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

Table of Contents

Preface

Developing applications alone is hard already, implementing applications using platforms that promise everything but turn out to be heavy-weight, hard to control and not very efficient during development cycles makes it even harder. Spring provides a light-weight solution to build enterprise-ready applications, still supporting the possibility of having declarative transaction management, remote access to your logic using RMI or webservices, mailing facilities and a decent way of persisting your data in a database. Spring provides an MVC framework, transparent ways of integrating AOP in your software and a well-structured exception hierarchy for for example JDBC.

Spring could potentially be a one-stop-shop for all your enterprise applications, however, Spring is modular, allowing you to use parts of it, without bothering about the rest. You can use the bean container, with Struts on top, but you could also choose to just use the Hibernate integration or the JDBC abstraction layer. Spring is non-intrusive, meaning you can do without any dependencies on the framework at all, thus not tying yourself in to Spring.

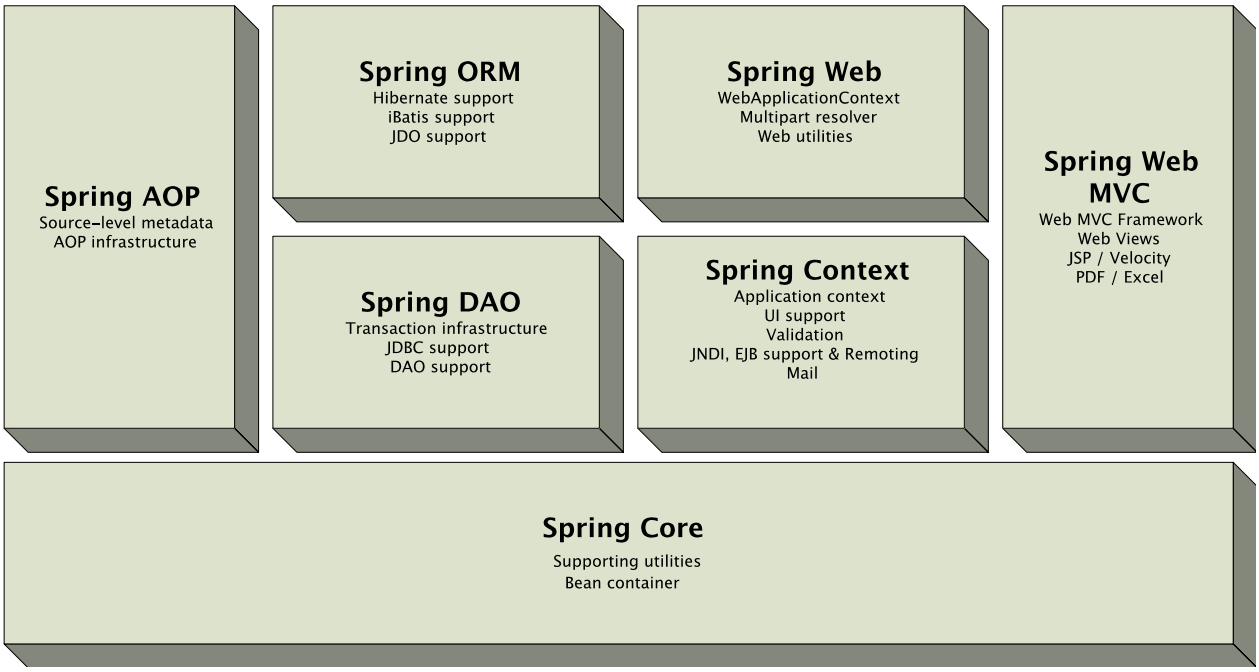
This document provides a reference guide to Spring's features. Since this document is still work-in-progress, if you have any requests, comments, please post them on the userlist or on the forum at the SourceForge project page: http://www.sf.net/projects/springframework

Before we go on a couple some words of gratitude. Chris Bauer (Hibernate) prepared and adapted the DocBook-XSL software in order to be able to create Hibernate's reference guide, also allowing us to create this one.

Chapter 1. Introduction

1.1. Overview

Spring contains a lot of functionality and features, which are well-organized in seven modules shown in the diagram below. This section discusses each the of modules in turn.



Overview of the the Spring Framework

The *Core* package is the most fundamental part of the framework and provides the Dependency Injection features allowing you to manage beans containing functionality. The basic concept here is the BeanFactory, which provides a factory pattern removing the need for programmatic singletons and allowing you to decouple the configuration and specification of dependencies from your actual logic.

On top of the *Core* package sits the *Context* package, providing a way to access beans in a framework-style manner, somewhat resembling a JNDI-registry. The context package inherits its features from the beans package and adds support text messaging using e.g. resource bundles, event-propagation, resource-loading and transparent creation of contexts by, for example, a servlet container.

The *DAO* package provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing database-vendor specific error codes. Also, the JDBC package provides a way to do programmatic as well as declarative transaction management, not only for classes implementing special interfaces, but for *all your POJOs*.

The *ORM* package provides integration layers for popular object-relational mappers, including JDO, Hibernate and iBatis. Using the ORM package you can use all those O/R-mappers in combination with all the other features Spring offers, like simple declarative transaction management mentioned before.

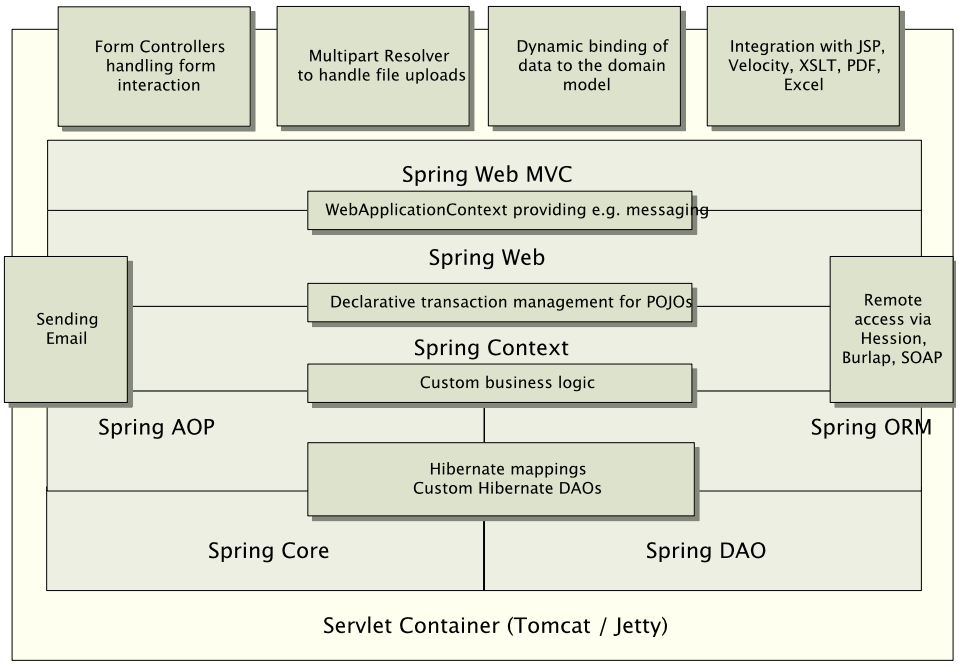
Spring's *AOP* package provides an *Aop Alliance* compliant aspect-oriented programming implementing allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated. Using source-level metadata functionality you can incorporate all kinds of behavioral information in your code, a little like .NET attributes.

Spring's *Web* package provides the basic web-oriented integration features, like multipart functionality, initialization of contexts using servlet listeners and a web-oriented application context. When using Spring together with WebWork or Struts, this is the package to integrate with.

Spring *Web MVC* package provides a Model-View-Controller implementation for web-applications. Spring's MVC implementation is not just any implementation, it provides a clean separation between domain model code and web forms and allows you to use all the other features of the Spring Framework like validation.

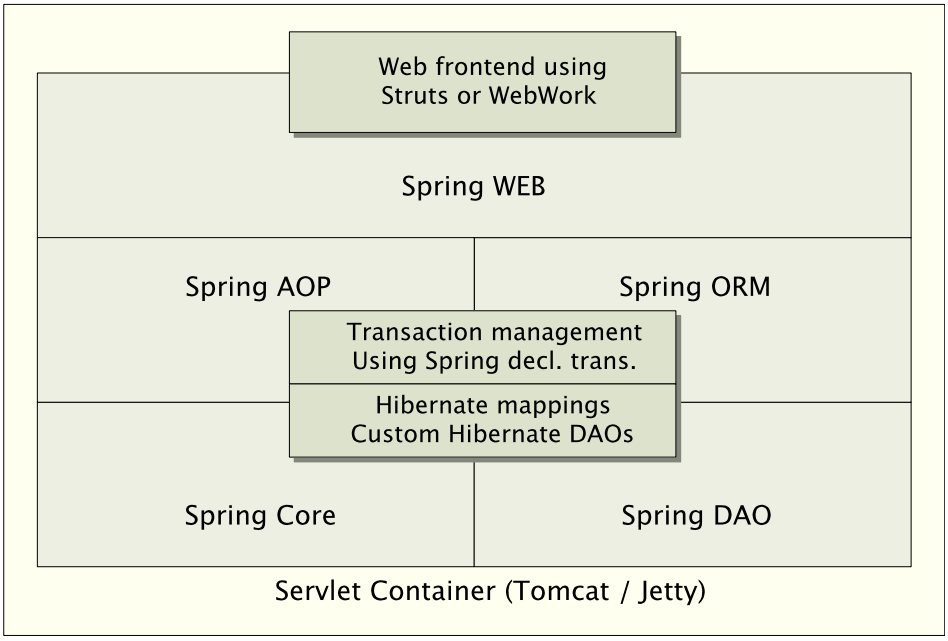
1.2. Usage scenarios

With the building block described above you can use Spring in all sorts of scenarios, from applets up to fully-fledged enterprise application using Spring's transaction management functionality and for example Spring's Web framework.



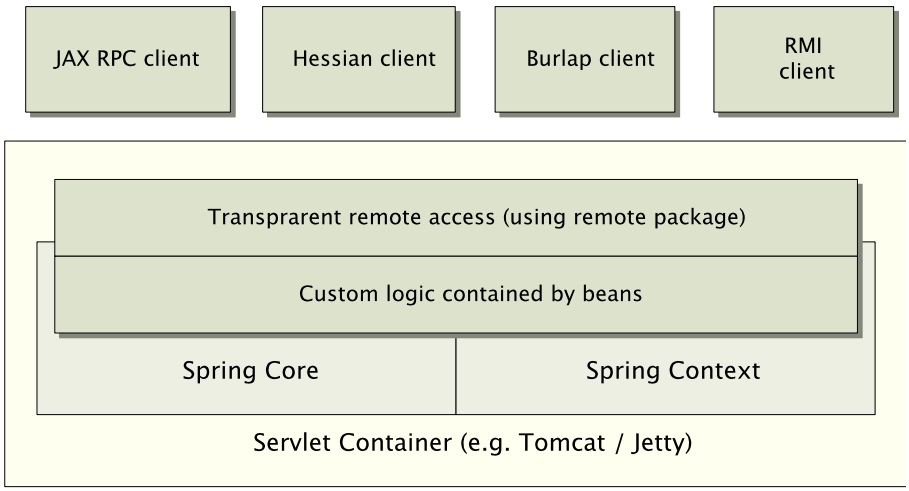
Typical full-fledged Spring web application

A typical web application using most of Spring's features. Using `TransactionProxyFactoryBeans` the web application is fully transactional, just as it would be when using container managed transaction as provided by Enterprise JavaBeans. All your custom business logic can be implemented using simple POJOs, managed by Spring's Dependency Injection container. Additional services like sending email and validation, independent of the web-layer enable you to choose where to execute validation rules. Spring's ORM support is integrated with Hibernate, JDO and iBatis. Using for example `HibernateDaoSupport`, you can re-use your existing Hibernate mappings. Form controllers seamlessly integrate the web-layer with the domain model, removing the need for `ActionForms` or other classes that transform HTTP parameters to values for your domain model.



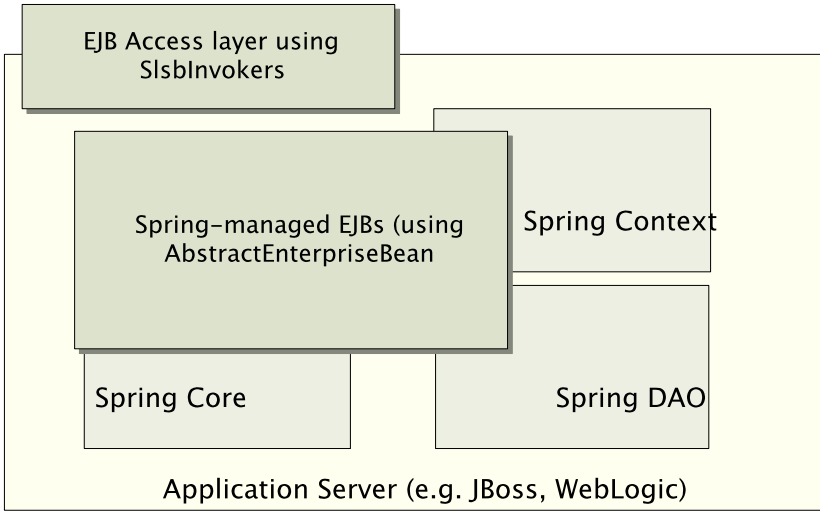
Spring middle-tier using a third-party web framework

Sometimes the situation does not allow you completely switch to a different framework. Spring does not enforce you to use everything, it's not an *all-or-nothing* solution. Existing frontends using WebWork or Struts can be integrated perfectly well with a Spring-based middle-tier, allowing you to use the transaction features that Spring offers. The only thing you need to do is wire up your business logic using an `ApplicationContext` and integrating your Struts frontend using a `WebApplicationContext`.



Remoting usage scenario

When you need to access existing code via webservices, you can use Spring's `Hessian-`, `Burlap-`, `Rmi-` or `JaxRpcProxyFactory` classes. Enabling remote access to existing application is all of a sudden not that hard anymore.



EJBs - Wrapping existing POJOs

Spring also provide an access-layer and abstraction-layer for Enterprise JavaBeans, enabling you to reuse your existing POJOs and wrap them in Stateless SessionBeans, for use in scalable failsafe webapplications, that might need declarative security.

Chapter 2. Background information

2.1. Inversion of Control / Dependency Injection

In early 2004, Martin Fowler asked the readers of his site: when talking about Inversion of Control: *"the question, is what aspect of control are they inverting?"*. After talking about the term Inversion of Control Martin suggests renaming the pattern, or at least giving it a more self-explanatory name, and starts to use the term *Dependency Injection*. His article continues to explain some of the ideas behind Inversion of Control or Dependency Injection. If you need a decent insight: http://martinfowler.com/articles/injection.html.

Chapter 3. Beans and the ApplicationContext

3.1. Introduction

Two of Spring's most elementary and important packages are the `org.springframework.beans` and `org.springframework.context` package. These packages provide the basis for the Spring Inversion of Control features, or Dependency Injection as people more recently call it (see Section 2.1, “Inversion of Control / Dependency Injection” for some resources on that). The `BeanFactory` provides an advanced configuration mechanism capable of managing beans of any kind of nature, using - potentially - any kind of storage facility. The `ApplicationContext` builds on top of the BeanFactory and adds other functionality such as integration with Springs AOP features, messaging (using *i18n*), the capability of having contexts inherit from other contexts and defining application-layer specific contexts, such as the `WebApplicationContext`.

In short, the `BeanFactory` provides the configuration framework and allows you to for example avoid the use of singletons, while the `ApplicationContext` adds more enterprise-centric, J2EE functionality to it.

This chapter is roughly divided into two parts, the first part covering the basic principles that apply to both the BeanFactory and the ApplicationContext. The second part will cover some of the features that only apply to the ApplicationContext.

In order to get you started using the BeanFactory and the ApplicationContext we'll first introduce you to a couple of basic principles that might be handy to know of when using the BeanFactory and the ApplicationContext.

- *Bean definition*: the description of a bean managed by the BeanFactory. Each BeanFactory contains bean definitions each describing the nature of a bean, its properties, the methods that need to be called upon initialization and destruction, whether or not the bean needs to be autowired and more things a like. The `XmlBeanFactory` for example is capalbe of reading bean definitions from an XML file, where properties, as well as collaborators and lifecycle methods are defined in XML
- *Property editors*: part of the JavaBeans specification by Sun is the concept of `PropertyEditors`. Property editors in Spring are used to convert properties of beans to human-readable String and vice versa. As the PropertyEditor JavaDOC (http://java.sun.com/j2se/1.4.2/docs/api/) states, the `getAsText()`-method should return a human-readable String representing the property the editor is (so-called) editing. The `setAsText()`-method should be able to parse the same String and (using the `setValue(Object)`-method) set the resulting value of the property. PropertyEditors are heavily used in Spring, not only by for example the BeanFactory to transform the Strings defined in the XML containing beans to properties of those beans, but also by the MVC framework to convert request parameters submitted by a user after filling a form to properties of command objects that might be of some completely different type
- *BeanWrapper*: an special class of which instances wrap a bean, managed BeanWrappers for example have features for setting and getting properties from the beans they're wrapping. It's not likely you'll ever have to touch a BeanWrapper yourself, but it might be handy to know about them

Note: there has been a lot of confusion about the use of the ApplicationContext as opposed to the BeanFactory. Basically when building applications in a J2EE-environment, the better option would be to use the ApplicationContext, since it offers a more framework style usage of the Spring Framework and can be initialized using a ServletContextListener like the `ContextLoaderListener`.

3.2. BeanFactory and BeanDefinitions - the basics

3.2.1. The BeanDefinition

As already stated in the introduction, bean definitions describe beans managed by a `BeanFactory` or `ApplicationContext`. Bean definitions contain the following information:

- 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), or the class that should be used to create the connection pool
- Other beans your bean needs to do its work, i.e. *collaborators* (also specified as properties or as constructor arguments)

In the list above, we mentioned the use of constructor arguments, as well as setters. Spring support two types of IoC, i.e. type 2 and type 3. What that means is that you use both constructor arguments to specify your dependencies and properties as well as setters and getters.

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 3.1. Bean definition explanation

Feature	**More info**
class	Section 3.2.3, “The bean class”
singleton or prototype	Section 3.2.5, “To singleton or not to singleton”
bean properties	Section 3.3.1, “Setting beans properties and collaborators”
constructor arguments	Section 3.3.1, “Setting beans properties and collaborators”
autowiring mode	Section 3.3.4, “Autowiring collaborators”
dependency checking mode	Section 3.3.5, “Checking for dependencies”
initialization method	Section 3.4.1, “Lifecycle interfaces”
destruction method	Section 3.4.1, “Lifecycle interfaces”

3.2.2. The BeanFactory

The BeanFactory is the actual *container* (although we don't like to use the word container too much, since it kind of sounds quite heavy), containing and managing your beans. A BeanFactory loads BeanDefinitions and - upon request - instantiates one (or possibly more) instances of the bean in question and manages it by calling lifecycle methods.

All of the features described in Section 3.2.1, “The BeanDefinition” will be configurable for each of your beans using one of the out-of-the-box BeanFactory implementations. In the example below we will use the most popular BeanFactory to illustrate the behavior, i.e. the `XmlBeanFactory`.

3.2.3. 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 are 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 connection pool that absolutely does not adhere to the bean specification, no worries, Spring can manage it as well.

Using the XmlBeanFactory you can specify your bean class as follows:

```
<bean id="exampleBean" class="examples.ExampleBean"/>
<bean name="anotherExample" class="examples.ExampleBeanTwo"/>
```

3.2.4. The bean identifiers (`id` and `name`)

Somehow of course you need to identifiy your beans. This can be done by a name or an id. The `id` is to be uniquely defined across all beans in a BeanFactory.

3.2.5. 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.

In the example below, two beans are declared of which one is defined as a singleton, and the other one as a prototype. `exampleBean` is created each and every time a client asks the BeanFactory for this bean, while `yetAnotherExample` is only created one, a reference is returned each time a request for this bean is done.

```
<bean id="exampleBean" class="examples.ExampleBean" singleton="false"/>
<bean name="yetAnotherExample" class="examples.ExampleBeanTwo" singleton="true"/>
```

3.3. Properties, collaborators, autowiring and dependency checking

3.3.1. Setting beans properties and collaborators

Inversion of Control has already been referred to as *Dependency Injection*. The basic principle is that beans themselves do not define who or what they're depending on, but instead, let the container do that for them.

Also, the container *injects* the actual dependencies, as opposed to the Service Locator pattern, where the beans themselves do a lookup to resolve those dependencies. While not elaborating too much on the advantages of Dependency Injection, it might be obvious that code gets much cleaner and reaching a higher grade of decoupling is much easier when beans themselves do not lookup their dependencies, but also do not even know where the actual dependencies are located and of what actual instance they are.

The BeanFactory is capable of *injecting* dependencies into beans its manages. It does so using the `BeanDefinition` and the `PropertyEditors` that are defined in the Spring Framework. We'll explain that further along using some examples of beans defined in the XML format. For now it might be handy to know that BeanDefinitions, collaborators and properties are specified using `PropertyValue`-objects.

The resolving of the dependencies is a little too complex to go in depth here, but 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, a Collection - i.e. Map or List, a Class or any other type Spring supports by default. It can also be a collaborator). Collaborators are other beans the BeanFactory must capable of resolving, in order words, other beans, also defined in the same BeanFactory (or, in case you're using the ApplicationContext, possibly in another application context)
2. In case the bean isn't a collaborator that can be resolved from the BeanFactory, Spring uses its built-in (or manually added) PropertyEditors to transform the property to the type that was required (when using the XML format to define a bean that contains a setter with a `Class`-parameter, Spring uses the `ClassEditor` (more about PropertyEditors and how to manually add custom ones, can be in Section 3.13, “Registering additional custom editors”.
3. In the case of collaborators, Spring constructs a reference to the bean that is used later on when the bean is actually instantiated. You can trust Spring here that it sets the actual collaborator at the right moments.

Dependency Injection (or Inversion of Control) exists in two major variants, both of which Spring supports.

- *setter-based* dependency injection is realized by calling setters on your beans after invoking an argumentless constructor to instantiate your bean. Beans defined in the BeanFactory that use setter-based dependency injection are *true JavaBeans*, and not only for the sake of adhering a standard, Spring advocated to use setter-based dependency injection as much as possible
- *constructor-based* dependency injection is realized by invoking a constructor with a number of arguments, each representing a collaborator or property. Though Spring advices to use setter-based dependency injection as much as possible, there might be beans around that just have constructors and that you want to use as well. Therefore we provided the constructor-based approach as well

Some examples:

First, an example of using the BeanFactory for setter-based dependency injection. 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 see, 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;
    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`.

3.3.2. Setting *null* values

Spring treats empty arguments for properties and the like as empty Strings. Consider the following code examples if you need to set *null*-values.

```
<bean class="ExampleBean">
    <property name="email"><value></value></property>
</bean>
```

results in `exampleBean.setEmail("");` as opposed to

```
<bean class="ExampleBean">
    <property name="email"><null/></property>
</bean>
```

which will result in: `exampleBean.setEmail(null)`.

3.3.3. Using `depends-on`

In normal situations, Spring resolves references to bean using for instance the `<ref bean/>`-tag. When just using those tags, without having special needs for initialization, you don't have to use the `depends-on`-tag. However, when you're using statics (which you of course shouldn't ;-) that need initialization or some beans needs to be initialized because of something else needing preparation, you can use the `depends-on`-tag, which will ensure all beans you're mentioning there, will get initialized before their actually set on the bean. For example:

```
<bean id="beanOne" class="ExampleBean" depends-on="manager">
    <property name="manager"><ref local="manager"/>
</bean>

<bean id="manager" class="ManagerBean"/>
```

3.3.4. 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[1]

Table 3.2. 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 type 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 applications, it is discouraged to use autowiring because it removes the transparency and the structure from your collaborating classes.

3.3.5. 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.

[1]See Section 3.3.1, “Setting beans properties and collaborators”

Table 3.3. Dependency checking modes

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

3.4. Customizing the nature of a bean

3.4.1. Lifecycle interfaces

Spring provides a couple of marker interfaces to change the behavior of your bean in the BeanFactory. They include `InitializingBean` and `DisposableBean`. Implementing those will result in the BeanFactory calling `afterPropertiesSet()` for the former and `destroy()` for the latter to allow you to do things upon initialization and destruction.

Internally, Spring uses `BeanPostProcessors` to process any marker interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring doesn't offer out-of-the-box, you can implement a `BeanFactoryPostProcessor` yourself. More information about this can be found in Section 3.6.1, “The BeanFactoryPostProcessor”.

All the different lifecycle marker interfaces are described below. In one of the appendices, you can find diagram as to how Spring manages beans and how those lifecycle features change the nature of your beans and how they are managed.

3.4.1.1. InitializingBean / `init-method`

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 to see if all necessary properties have been set correctly, or to perform further 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).

```
<bean id="exampleInitBean" class="examples.ExampleBean" init-method="init"/>

public class ExampleBean {
    public void init() {
        // do some initialization work
    }
}
```

Is exactly the same as:

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements InitializingBean {
    public void afterPropertiesSet() {
        // do some initialization work
    }
}
```

3.4.1.2. DisposableBean / `destroy-method`

The `org.springframework.beans.factory.DisposableBean` interface provides you with the ability to get a callback when a beanfactory 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)

```
<bean id="exampleInitBean" class="examples.ExampleBean" destroy-method="cleanup()"/>

public class ExampleBean {
    public void cleanup() {
        // do some destruction work (like closing connection)
    }
}
```

Is exactly the same as:

```
<bean id="exampleInitBean" class="examples.AnotherExampleBean"/>

public class AnotherExampleBean implements DisposableBean {
    public void destroy() {
        // do some destruction work
    }
}
```

3.4.2. Knowing who you are

3.4.2.1. 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).

3.4.2.2. BeanNameAware

The `org.springframework.beans.factory.BeanNameAware` interface gives you the ability to let the BeanFactory set the name of the bean on the bean itself. In case you need to know what your name is,

implement this interface

- `void setBeanName(String)`: method which will be called to let the bean know what its name is

3.4.3. FactoryBean

The `org.springframework.beans.factory.FactoryBean` is to be implemented by 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

3.5. Interacting with the BeanFactory

Basically the BeanFactory is nothing more than an advanced factory capable of maintaining a registry oof different components or beans. The BeanFactory enables you to read bean definitions and access them using the bean factory. When just using the BeanFactory you would create one and read in some bean definitions in the XML format as follows:

```
InputStream is = new FileInputStream("beans.xml");
XmlBeanFactory factory = new XmlBeanFactory(is);
```

Basically that's all there is to it. Using `getBean(String)` you can retrieve instances of your beans. You'll get a reference to the same bean if you defined it as a singleton (which is the default) and you'll get a new instance each time if you set `singleton` to *false*. 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 case 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 corresponding to the given name could not be found, an exception will be thrown (`NoSuchBeanDefinitionException`)
- `String[] getAliases(String)`: returns the aliases configured for this bean (TODO: WHAT IS THIS :)

3.6. Customizing the BeanFactory

When in need for special behavior for the BeanFactory you need to consult the `org.springframework.beans.factory.config`-package, which should provide you all the things you need in such a case. Using the config package you can for instance define a properties file of which the property-entries are used as replacements values in the BeanFactory. Also, you can implement custom behavior (like adding custom editors) using the BeanFactoryPostProcessor. Each of the features the config-package offers is

described below:

3.6.1. The `BeanFactoryPostProcessor`

The BeanFactoryPostProcessor is the superclass for all post processors that are capable of applying changes to the BeanFactory (or beans in it) after the factory has been initialized. The BeanFactoryPostProcessor can be used to add custom editors (as also mentioned in Section 4.3.2, “Built-in PropertyEditors, converting types”). Some out-of-the-box post processors are available, like the `PropertyResourceConfigurer` and the `PropertyPlaceHolderConfigurer`, both described below, and `BeanNameAutoProxyCreator`, very useful for wrapping other beans transactionally, as described later in this manual.

3.6.2. The `PropertyPlaceholderConfigurer`

The `PropertyPlaceholderConfigurer` is an excellent solution when you want to externalize a few properties from a file containing bean definitions. Say you want to let the person responsible for deploying applications just fill in database URLs, usernames and passwords. You could of course let him change the XML file containing bean definitions, but that would be risky. Instead, use the `PropertyPlaceHolderConfigurer` to - at runtime - replace those properties by values from a properties file.

In the example below, a datasource is defined, and we will configure some properties from an external Properties file. At runtime, we will apply a `PropertyPlaceholderConfigurer` to the BeanFactory which will replace some properties of the datasource:

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName"><value>${jdbc.driverClassName}</value></property>
    <property name="url"><value>${jdbc.url}</value></property>
    <property name="username"><value>${jdbc.username}</value></property>
    <property name="password"><value>${jdbc.password}</value></property>
</bean>
```

```
jdbc.driverClassName=org.hsqldb.jdbcDriver
jdbc.url=jdbc:hsqldb:hsql://production:9002
jdbc.username=sa
jdbc.password=root
```

```
XmlBeanFactory factory = new XmlBeanFactory(new FileInputStream("beans.xml"));

Properties props = new Properties();
props.load(new FileInputStream("jdbc.properties"));

PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer(props);
cfg.postProcessBeanFactory(factory);
```

The `PropertyPlaceHolderConfigurer` does not only look for properties in the properties file you're defining, but also checks against the System properties if it cannot find the properties you're trying to replace. This behavior can be customized by tweaking with the `systemPropertiesMode` property of the configurer. It has three values, one to tell the configurer to always override, one to let it *never* override and one to let it override only if the property cannot be found in the properties file specified. Please consult the JavaDoc for the PropertiesPlaceHolderConfigurer for more information.

3.7. Lifecycle of a bean in the BeanFactory

This chapter will be re-written soon.

3.8. BeanFactory structure and implementations (WIP)

From here on, up till the Context stuff, needs an overhaul.

The `beans` package contains a couple of different implementations. Concrete BeanFactories include the XmlBeanFactory which is the most popular format at the moment. However, the `ListableBeanFactory` is also a nice one, which is capable of reading bean definitions from properties files.

3.8.1. Structure of the beans package

Include UML and description here

3.8.2. The `XmlBeanFactory`

The `XmlBeanFactory` is the most often-used implementation of the BeanFactory and is capable of reading bean definitions from XML files. XML files need to adhere to the Spring DTD, included in Appendix

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.

3.8.3. 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.

3.9. Introduction to the `ApplicationContext`

The `context` package is built on top of the `beans` package which provides basic functionality for managing and manipulating beans, often in a programmatic way. The `ApplicationContext` introduces BeanFactory functionality in a more *framework-style* approach. Instead of programmatically loading beans using for example the XmlBeanFactory and retrieving them using the `getBean()` method, with the ApplicationContext you will be able to let the framework you're working in load your beans using for example the `ContextLoader`.

The basis for the context package is the `ApplicationContext` interface, located in the `org.springframework.context` package. First of all it provides all the functionality the BeanFactory also provides. To be able to work in a more framework-oriented way, using layering and hierarchical contexts, the context package also provides the following:

- *MessageSource*, providing access to messages in, i18n-style
- *Access to resources*, like URLs and files
- *Event propagation* to beans implementing the `ApplicationListener` interface
- *Loading of multiple contexts*, allowing some of them to be focused and used for example only by the web-layer of an application

The ApplicationContext includes all the functionality the bean factory has as well. This means you can define beans using the XML format as explained in Section 3.8.2, “The XmlBeanFactory” and Appendix A, *Spring's beans.dtd* and retrieve them using their name or id. Also, you will be able to use all of the other features explained in this chapter, such as *autowring*, *setter-based* or *constructor-based* depenency injection, *dependency checking*, *lifecycle interfaces*, etcetera. All other features the ApplicationContext includes are described below, as well as the ways to access an ApplicationContext, load beans into an ApplicationContext and the ways to customize existing ApplicationContexts using special beans defined in them.

3.10. Added functionality of the `ApplicationContext`

As already stated in the previous section, the ApplicationContext has a couple of features that distinguish it from the BeanFactory. Let us review them one-by-one.

3.10.1. Using the `MessageSource`

The ApplicationContext interface extends an interface called `MessageSource`, and therefore provides messaging (i18n) functionality. Together with the `NestingMessageSource`, capable of hierarchical message resolving, these are the basic interfaces Spring provides to do message resolving. Let's quickly review the methods defined there:

- `String getMessage (String code, Object[] args, String default, Locale loc)`: the basic method to retrieve a message from the MessageSource. When no message could be found for the locale specified, the default message is used. Any arguments passed in are used as replacement values, using the `MessageFormat` functionality provided by the JDK
- `String getMessage (String code, Object[] args, Locale loc)`: basically the same as the one above

with one difference: the fact that no default message can be specified results in a `NoSuchMessageException` being thrown if the message could not be found

- `String getMessage(MessageSourceResolvable resolvable, Locale locale)`: all properties used in the methods above are also wrapped in a class - the `MessageSourceResolvable`, which you can use in this method

When an ApplicationContext gets loaded, it automatically searches for a MessageSource bean defined in the context. The bean has to have the name `messageSource`. If such a bean is found, all calls to the methods desribed above will be delegated to the message source that was found. If no message source was found, the ApplicationContext inspects to see if it has a parent containing a similar bean, with a similar name. If so, it uses that bean as the MessageSource. If it can't find any source for messages, an empty `StaticMessageSource` will be instantiated in order to be able to accept calls to the methods defined above.

Spring provides two `MessageSource` implementation at the moment. These are the `ResourceBundleMessageSource` and the `StaticMessageSource`. Both implement `NestingMessageSource` in order to be able to do nested messaging. The StaticMessageSource is hardly ever used and provides programmatic ways to add messages to the source. The ResourceBundleMessageSource is more interesting and is the one we will provides an example for:

```
<beans>
    <bean id="messageSource"
            class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="baseNames">
            <list>
                <value>format</value>
                <value>exceptions</value>
                <value>windows</value>
            </list>
        </property>
    </bean>
</beans>
```

This assumes you have three resource bundles defined on your classpath called `format`, `exceptions` and `windows`. Using the JDK standard way of resolving messages through ResourceBundles, any request to resolve a message will be handled.

3.10.2. Propagating events

Eventhandling in the ApplicationContext is provided through the `ApplicationEvent` class and `ApplicationListener` interface. When implementing the `ApplicationListener` in one of your beans, everytime an `ApplicationEvent` gets published to the ApplicationContext, your bean will be notified, based on the standard Observer design pattern implemented by the `java.util` package. Spring provides three standard events:

Table 3.4. Built-in Events

Event	Explanation
`ContextRefreshedEvent`	Event published when the ApplicationContext is initialized or refreshed. Initialized here means that all beans are loaded, singletons are pre-instantiated and the ApplicationContext is ready for use
`ContextClosedEvent`	Event published when the ApplicationContext is closed, using the `close()` method on the ApplicationContext. Closed here means that are singletons are destroyed
`RequestHandledEvent`	A web-specific event telling all beans that a HTTP request has been serviced

Event	Explanation
	(so it'll be published *after* the request has been finished). Note that this event is only applicable for web applications using Spring's DispatcherServlet

Implementing custom events can be done as well. All you need to do is call the `publishEvent()` method on the ApplicationContext and you're done. Let's have a look at an example. First, the ApplicationContext:

```
<bean id="emailer" class="example.EmailBean">
    <property name="blackList">
        <list>
            <value>black@list.org</value>
            <value>white@list.org</value>
            <value>john@doe.org</value>
        </list>
    </property>
</bean>

<bean id="blackListListener" class="example.BlackListNotifier">
    <property name="notificationAddress">
        <value>spam@list.org</value>
    </property>
</bean>
```

and then, the actual beans:

```
public class EmailBean implements ApplicationContextAware {

    /** the blacklist */
    private List blackList;

    public void setBlackList(List blackList) {
        this.blackList = blackList;
    }

    public void setApplicationContext(ApplicationContext ctx) {
        this.ctx = ctx;
    }

    public void sendEmail(String address, String text) {
        if (blackList.contains(address)) {
            BlackListEvent evt = new BlackListEvent(address, text);
            ctx.publishEvent(evt);
            return;
        }
        // send email
    }
}

public class BlackListNotifier implement ApplicationListener {

    /** notification address */
    private String notificationAddress;

    public void setNotificationAddress(String notificationAddress) {
        this.notificationAddress = notificationAddress;
    }

    public void onApplicationEvent(ApplicationEvent evt) {
        if (evt instanceof BlackListEvent) {
            // notify appropriate person
        }
    }
}
```

Of course, this behavior could be more cleanly implemented maybe, by using AOP features, but just to illustrate the basic behavior, this should do.

3.10.3. Using resources within Spring

Emphasis: not sure where to cover this, let's leave it for now (so it's not complete yet).

A lot of applications need to access resources. Resources here, might mean files, but also newsfeeds from the Internet or normal webpages. Spring provides a clean and transparent way of accessing resources in a protocol independent way. The ApplicationContext has a method (`getResource(String)`) to take care of this.

The Resource class defines a couple of methods that are shared across all Resource implementations:

Table 3.5. Resource functionality

Method	Explanation
`getInputStream()`	Opens an InputStream on the resource and returns it
`exists()`	Checks if the resource exists, returning false if it doesn't
`isOpen()`	Will return true is multiple streams cannot be opened for this resource. This will be false for some resources, but file-based resources for instance, cannot be read multiple times concurrently
`getDescription()`	Returns a description of the resource, often the fully qualified file name or the actual URL

A couple of Resource implementations are provided by Spring. They all need a String representing the actual location of the resource. Based upon that String, Spring will automatically choose the right Resource implementation for you. When asking an ApplicationContext for a resource first of all Spring will inspect the resource location you're specifying and look for any prefixes. Depending on the implenentation of the ApplicationContext more or less Resource implementations are available. Resources can best be configured by using the ResourceEditor and for example the XmlBeanFactory.

3.11. Extra marker interfaces and lifecycle features

The BeanFactory already offers you some methods to control the lifecycle of your beans (like `InitializingBean` or `init-method` and its counterpart, `DisposableBean` or `destroy-method`). The `context` package builds on top of the BeanFactory so you can use those features in the context package as well. Besides the lifecycle features, the ApplicationContext has one extra marker interface you can add to your beans, the `ApplicationContextAware` interface. This interface will be used to let the bean know about the ApplicationContext it is *contained* in, using the `setApplicationContext()` method, which provides the ApplicationContext as an argument.

3.12. Customization of the ApplicationContext

Where the BeanFactory can be customized programmatically using `BeanFactoryPostProcessor`, the ApplicationContext can be customized using special beans, defined in your ApplicationContext

3.13. Registering additional custom editors

When in need of a custom editor in your BeanFactory or ApplicationContext (e.g. you have some exotic type that you need to express as a property in a BeanFactory XML file), you can use the `CustomEditorConfigurer`,

which is a special kind of `BeanFactoryPostProcessor` (more about those can be found in Section 3.6, “Customizing the BeanFactory”). Consider the following code samples:

```
public class ExoticType {

    private String name;

    public ExoticType(String name) {
        this.name = name;
    }
}

public class DependsOnExoticType {

    private ExoticType type;

    public void setType(ExoticType type) {
        this.type = type;
    }
}
```

Wiring this up should be done something like this:

```
<bean id="sample" class="example.DependsOnExoticType">
    <property name="type"><value>exoticTypeTwo</value></property>
</bean>
```

When you want to do this, you would need to create a PropertyEditor and wire it up using the `CustomEditorConfigurer`:

```
public class ExoticTypeEditor extends PropertyEditorSupport {

    private String format;

    public void setWhatever(String format) {
        this.format = format

    public void setAsText(String text) {
        if (format != null && format.equals("upperCase")) {
            text = text.toUpperCase();
        }
        ExoticType type = new ExoticType(text);
        setValue(type);
    }
}
```

To actually register it, do the following:

```
<bean id="customEditorConfigurer"
    class="org.springframework.bean.factory.config.CustomEditorConfigurer">
    <property name="customEditors">
        <map>
            <entry key="example.ExoticType">
                <bean class="example.ExoticTypeEditor">
                    <property name="format">
                        <value>upperCase</value>
                    </property>
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

Besides using the `CustomEditorConfigurer`, you can also register custom editors programmatically if you need. This might come in handy when for instance using the BeanFactory (which does not automatically detect BeanFactoryPostProcessors). Use the `ConfigurableBeanFactory.registerCustomEditor()` method to do this.

3.14. Setting a bean property as the result of a method

invocation

It is sometimes necessary to set a property on a bean, as the result of a method call on another bean in the container, or a static method call on any arbitrary class. Additionally, it is sometimes necessary to call a static or non-static method just to perform some sort of initialization. For both of these purposes, a helper class called `MethodInvokingFactoryBean` may be used. This is a `FactoryBean` which returns a value which is the result of a static or instance method invocation.

An example (in an XML based BeanFactory definition) of a bean definition which uses this class to call a static factory method:

```
<bean id="myClass" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="staticMethod"><value>com.whatever.MyClassFactory.getInstance</value></property>
</bean>
```

An example of calling a static method then an instance method to get at a Java System property. Somewhat verbose, but it works.

```
<bean id="sysProps" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetClass"><value>java.lang.System</value></property>
  <property name="targetMethod"><value>getProperties</value></property>
</bean>
<bean id="javaVersion" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetObject"><ref local='sysProps'/></property>
  <property name="targetMethod"><value>getProperty</value></property>
  <property name="args">
    <list>
      <value>|java.version|</value>
    </list>
  </property>
</bean>
```

Note that as it is expected to be used mostly for accessing factory methods, MethodInvokingFactoryBean by default operates in a *singleton* fashion. The first request by the container for the factory to produce an object will cause the specified method invocation, whose return value will cached and returned for the current and subsequent requests. An internal singleton property of the factory may be set to false, to cause it to invoke the target method each time it is asked for an object.

A static target method may be specified by setting targetMethod property to a String representing the static method name, with targetClass specifying the Class that the static method is defined on. Alternatively, a target instance method may be specified, by setting the targetObject property as the target object, and the targetMethod property as the name of the method to call on that target object. Arguments for the method invocation may be specified by setting the args property.

3.15. Creating an ApplicationContext from a web application

As opposed to the BeanFactory, which will often be created programmatically, ApplicationContexts can be created declaratively using for example a `ContextLoader`. Of course you can also create ApplicationContexts programmatically using one of the ApplicationContext implementations. First, let's examine the ContextLoader and its implementations.

The ContextLoader has two implementations: the `ContextLoaderListener` and the `ContextLoaderServlet`. They both have the same functionality but differ in that the listener cannot be used in Servlet 2.2 compatible containers. Since the Servlet 2.4 specification, listener are required to initialize after startup of a

webapplication. A lot of 2.3 compatible containers already implement this feature. It is up to you as to which one you use, but all things being equal you should probably prefer `ContextLoaderListener`; for more information on compatibility, have a look at the JavaDoc for the `ContextLoaderServlet`.

You can register an ApplicationContext using the `ContextLoaderListener` as follows:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/daoContext.xml /WEB-INF/applicationContext.xml</param-value>
</context-param>

<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<!-- OR USE THE CONTEXTLOADERSERVLET INSTEAD OF THE LISTENER
<servlet>
    <servlet-name>context</servlet-name>
    <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
-->
```

The listener inspects the `contextConfigLocation` parameter. If it doesn't exist, it'll use `/WEB-INF/applicationContext.xml` as a default. When it *does* exist, it'll separate the String using predefined delimiters (comma, semi-colon and space) and use the values as locations where application contexts will be searched for. The ContextLoaderServlet can - as said - be used instead of the ContextLoaderListener. The servlet will use the contextConfigLocation parameter just as the listener does.

3.16. Glue code and the evil singleton

The majority of the code inside an application is best written in a Dependency Injection (Inversion of Control) style, where that code is served out of a BeanFactory or ApplicationContext container, has its own dependencies supplied by the container when it is created, and is completely unaware of the container. However, for the small glue layers of code that are sometimes needed to tie other code together, there is sometimes a need for singleton (or quasi-singleton) style access to a BeanFactory or ApplicationContext. For example, third party code may try to construct new objects directly (`Class.forName()` style), without the ability to force it to get these objects out of a BeanFactory. If the object constructed by the third party code is just a small stub or proxy, which then uses a singleton style access to a BeanFactory/ApplicationContext to get a real object from, to which object it then delegates, then inversion of control has still been achieved for the majority of the code (the object coming out of the BeanFactory); thus most code is still unaware of the container or how it is accessed, and remains uncoupled from other code, with all ensuing benefits. EJBs may also use this stub/proxy approach to delegate to a plain java implementation object, coming out of a BeanFactory. While the BeanFactory ideally does not have to be a singleton, it may be unrealistic in terms of memory usage or initialization times (when using beans in the BeanFactory such as a Hibernate SessionFactory) for each bean to use its own, non-singleton BeanFactory.

As another example, in a complex J2EE apps with multiple layers (i.e. various JAR files, EJBs, and WAR files packaged as an EAR), with each layer having its own ApplicationContext definition (effectively forming a hierarchy), the preferred approach when there is only one web-app (WAR) in the top hierarchy is to simply create one composite ApplicationContext from the multiple XML definition files from each layer. All the ApplicationContext variants are able to be constructed from multiple definition files in this fashion. However, if there are multiple sibling web-apps at the top of the hierarchy, it is problematic to create an ApplicationContext for each web-app which conssists of mostly identical bean definitions from lower layers, as there may be issues due to increased memory usage, issues with creating mutliple copies of beans which take a long time to initialize (i.e. a Hibernate SessionFactory), and possible issues due to side-effects. As an alternative, classes like `ContextSingletonBeanFactoryLocator` or `SingletonBeanFactoryLocator` may be

used to demand load multiple hierarchical (i.e. one is a parent of another) BeanFactories or ApplicationContexts in an effectively singleton fashion, which may then be used as the parents of the web-app ApplicationContexts. The result is that bean definitions for lower layers are loaded only as needed, and loaded only once.

3.16.1. Using SingletonBeanFactoryLocator and ContextSingletonBeanFactoryLocator

You can see a detailed example of using `SingletonBeanFactoryLocator` and `ContextSingletonBeanFactoryLocator` by viewing their respective JavaDocs, available at http://www.springframework.org/docs/api/org/springframework/beans/factory/access/SingletonBeanFactoryLocator.html and http://www.springframework.org/docs/api/org/springframework/context/access/ContextSingletonBeanFactoryLocator.html.

As mentioned in the chapter on EJBs, the Spring convenience base classes for EJBs normally use a non-singleton `BeanFactoryLocator` implementation, which is easily replaced by the use of `SingletonBeanFactoryLocator` and `ContextSingletonBeanFactoryLocator` if there is a need.

Chapter 4. PropertyEditors, data binding, validation and the BeanWrapper

4.1. Introduction

The big question is whether or not validation should be considered *business logic*. There are 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 but not limited to the MVC framework.

The `BeanWrapper` is a fundamental concept in the Spring Framework and is used in a lot of places. However, you probably will not ever have the need to use the BeanWrapper directly. Because this is reference documentation however, we felt that some explanation might be right. We're explaining the BeanWrapper in this chapter since if you were going to use it at all, you would probably do that when trying to bind data to objects, which is strongly related to the BeanWrapper.

Spring uses PropertyEditors all over the place. The concept of a PropertyEditor is part of the JavaBeans specification. Just as the BeanWrapper, it's best to explain the use of PropertyEditors in this chapter as well, since it's closely related to the BeanWrapper and the DataBinder.

4.2. Binding data using the `DataBinder`

The DataBinder builds on top of the BeanWrapper[2].

4.3. 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's 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 functionnality 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`.

[2]See the beans chapter for more information

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

4.3.1. Setting and getting basic and nested properties

Setting and getting properties is done using the `setPropertyValue(s)` and `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 is that there are a couple of conventions for indicating properties of an object. A couple of examples:

Table 4.1. 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 *naturally 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");
```

4.3.2. Built-in `PropertyEditors`, converting types

Spring heavily uses the concept of `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 on 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.

A couple of examples where property editing is used in Spring

- *setting properties on beans* is done using PropertyEditors. When mentioning `java.lang.String` as the value of a property of some bean you're declaring in XML file, Spring will (if the setter of the corresponding property has a Class-parameter) use the `ClassEditor` to try to resolve the parameter to a Class object
- *parsing HTTP request parameters* in Spring's MVC framework is done using all kinds of PropertyEditors that you can manually bind in all subclasses of the `CommandController`

Spring has a number of built-in PropertyEditors to make life easy. Each of those is listed below and they are all located in the `org.springframework.beans.propertyeditors` package:

Table 4.2. Built-in PropertyEditors

Class	Explanation
`ClassEditor`	Parses Strings representing classes to actual classes and the other way around. When a class is not found, an IllegalArgumentException is thrown
`FileEditor`	Capable of resolving String to `File`-objects
`LocaleEditor`	Capable of resolving Strings to `Locale`-objects and vice versa (the String format is [language]_[country]_[variant], which is the same thing the toString() method of Locale provides
`PropertiesEditor`	Capable of converting Strings (formatted using the format as defined in the JavaDOC for the java.lang.Properties class) to

Class	Explanation
	`Properties`-objects
`StringArrayPropertyEditor`	Capable of resolving a comma-delimited list of String to a String-array and vice versa
`URLEditor`	Capable of resolving a String representation of a URL to an actual `URL`-object

Spring uses the `java.beans.PropertyEditorManager` to set the search-path for property editors that might be needed. The search-path also includes `sun.bean.editors`, which includes PropertyEditors for Font, Color and all the primitive types.

4.3.3. Other features worth mentioning

Besides the features you've seen in the previous sections there a couple of features that might be interesting 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

Chapter 5. Spring AOP: Aspect Oriented Programming with Spring

5.1. Concepts

Aspect-Oriented Programming (*AOP*) complements OOP by providing another way of thinking about program structure. While OO decomposes applications into a hierarchy of objects, AOP decomposes programs into *aspects* or *concerns*. This enables modularization of concerns such as transaction management that would otherwise cut across multiple objects. (Such concerns are often termed *crosscutting* concerns.)

One of the key components of Spring is the *AOP framework*. While the Spring IoC containers (BeanFactory and ApplicationContext) do not depend on AOP, meaning you don't need to use AOP if you don't want to, AOP complements Spring IoC to provide a very capable middleware solution.

AOP is used in Spring:

- To provide declarative enterprise services, especially as a replacement for EJB declarative services. The most important such service is *declarative transaction management*, which builds on Spring's transaction abstraction.
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

Thus you can view Spring AOP as either an enabling technology that allows Spring to provide declarative transaction management without EJB; or use the full power of the Spring AOP framework to implement custom aspects.
If you are interested only in generic declarative services or other pre-packaged declarative middleware services such as pooling, you don't need to work directly with Spring AOP, and can skip most of this chapter.

5.1.1. AOP concepts

Let us begin by defining some central AOP concepts. These terms are not Spring-specific. Unfortunately, AOP terminology is not particularly intuitive. However, it would be even more confusing if Spring used its own terminology.

- *Aspect*: A modularization of a concern for which the implementation might otherwise cut across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. Aspects are implemented using Spring as Advisors or interceptors.
- *Joinpoint*: Point during the execution of a program, such as a method invocation or a particular exception being thrown.
- *Advice*: Action taken by the AOP framework at a particular joinpoint. Different types of advice include "around," "before" and "throws" advice. Advice types are discussed below. Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the joinpoint.
- *Pointcut*: A set of joinpoints specifying when an advice should fire. An AOP framework must allow developers to specify pointcuts: for example, using regular expressions.

- *Introduction*: Adding methods or fields to an advised class. Spring allows you to introduce new interfaces to any advised object. For example, you could use an introduction to make any object implement an `IsModified` interface, to simplify caching.
- *Target object*: Object containing the joinpoint. Also referred to as *advised* or *proxied* object.
- *AOP proxy*: Object created by the AOP framework, including advice. In Spring, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy.
- *Weaving*: Assembling aspects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), or at runtime. Spring, like other pure Java AOP frameworks, performs weaving at runtime.

Different advice types include:

- *Around advice*: Advice that surrounds a joinpoint such as a method invocation. This is the most powerful kind of advice. Around advices will perform custom behaviour before and after the method invocation. They are responsible for choosing whether to proceed to the joinpoint or to shortcut executing by returning their own return value or throwing an exception.
- *Before advice*: Advice that executes before a joinpoint, but which does not have the ability to prevent execution flow proceeding to the joinpoint (unless it throws an exception).
- *Throws advice*: Advice to be executed if a method throws an exception. Spring provides strongly typed throws advice, so you can write code that catches the exception (and subclasses) you're interested in, without needing to cast from Throwable or Exception.
- *After returning advice*: Advice to be executed after a joinpoint completes normally: for example, if a method returns without throwing an exception.

Around advice is the most general kind of advice. Most interception-based AOP frameworks, such as Nanning Aspects and JBoss 4 (as or DR2), provide only around advice.

As Spring, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behaviour. For example, if you need only to update a cache with the return value of a method, you are better off implementing an after returning advice than an around advice, although an around advice can accomplish the same thing. Using the most specific advice type provides a simpler programming model with less potential for errors. For example, you don't need to invoke the `proceed()` method on the MethodInvocation used for around advice, and hence can't fail to invoke it.

The pointcut concept is the key to AOP, distinguishing AOP from older technologies offering interception. Pointcuts enable advice to be targeted independently of the OO hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects. Thus pointcuts provide the structural element of AOP.

5.1.2. Spring AOP capabilities

Spring AOP is implemented in pure Java. There is no need for a special compilation process. Spring AOP does not need to control the class loader hierarchy, and is thus suitable for use in a J2EE web container or application server.

Spring currently supports interception of method invocations. Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs.

Field interception arguably violates OO encapsulation. We don't believe it is wise in application development. If you require field interception, consider using AspectJ.

Spring provides classes to represent pointcuts and different advice types. Spring uses the term *advisor* for an object representing an aspect, including both an advice and a pointcut targeting it to specific joinpoints.

Different advice types are `MethodInterceptor` (from the AOP Alliance interception API); and the advice interfaces defined in the `org.springframework.aop` package. All advices must implement the `org.aopalliance.aop.Advice` tag interface. Advices supported out the box are `MethodInterceptor` ; `ThrowsAdvice`; `BeforeAdvice`; and `AfterReturningAdvice`. We'll discuss advice types in detail below.

Spring implements the *AOP Alliance* interception interfaces (http://www.sourceforge.net/projects/aopalliance). Around advice must implement the AOP Alliance `org.aopalliance.intercept.MethodInterceptor` interface. Implementations of this interface can run in Spring or any other AOP Alliance compliant implementation. Currently JAC implements the AOP Alliance interfaces, and Nanning and Dynaop are likely to in early 2004.
Spring's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.
Thus, for example, Spring's AOP functionality is normally used in conjunction with a Spring IoC container. AOP advice is specified using normal bean definition syntax (although this allows powerful "autoproxying" capabilities); advice and pointcuts are themselves managed by Spring IoC: a crucial difference from other AOP implementations. There are some things you can't do easily or efficiently with Spring AOP, such as advise very fine-grained objects. AspectJ is probably the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in J2EE applications that are amenable to AOP.

5.1.3. AOP Proxies in Spring

Spring defaults to using JDK *dynamic proxies* for AOP proxies. This enables any interface or set of interfaces to be proxied.

Spring can also use CGLIB proxies. This is necessary to proxy classes, rather than interfaces. CGLIB is used by default if a business object doesn't implement an interface. As it's good practice to *program to interfaces rather than classes*, business objects normally will implement one or more business interfaces.

It is possible to force the use of CGLIB: we'll discuss this below, and explain why you'd want to do this.
Beyond Spring 1.0, Spring may offer additional types of AOP proxy, including wholly generated classes. This won't affect the programming model.

5.2. Pointcuts in Spring

Let's look at how Spring handles the crucial pointcut concept.

5.2.1. Concepts

Spring's pointcut model enables pointcut reuse independent of advice types. It's possible to target different advice using the same pointcut.

The `org.springframework.aop.Pointcut` interface is the central interface, used to target advices to particular classes and methods. The complete interface is shown below:

```
public interface Pointcut {
```

```
    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

Splitting the `Pointcut` interface into two parts allows reuse of class and method matching parts, and fine-grained composition operations (such as performing a "union" with another method matcher).

The `ClassFilter` interface is used to restrict the pointcut to a given set of target classes. If the `matches()` method always returns true, all target classes will be matched:

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

The `MethodMatcher` interface is normally more important. The complete interface is shown below:

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

    boolean matches(Method m, Class targetClass, Object[] args);
}
```

The `matches(Method, Class)` method is used to test whether this pointcut will ever match a given method on a target class. This evaluation can be performed when an AOP proxy is created, to avoid the need for a test on every method invocation. If the 2-argument matches method returns true for a given method, and the `isRuntime()` method for the MethodMatcher returns true, the 3-argument matches method will be invoked on every method invocation. This enables a pointcut to look at the arguments passed to the method invocation immediately before the target advice is to execute.

Most MethodMatchers are static, meaning that their `isRuntime()` method returns false. In this case, the 3-argument matches method will never be invoked.
If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

5.2.2. Operations on pointcuts

Spring supports operations on pointcuts: notably, *union* and *intersection*.

Union means the methods that either pointcut matches.

Intersection means the methods that both pointcuts match.

Union is usually more useful.

Pointcuts can be composed using the static methods in the *org.springframework.aop.support.Pointcuts* class, or using the *ComposablePointcut* class in the same package.

5.2.3. Convenience pointcut implementations

Spring provides several convenient pointcut implementations. Some can be used out of the box; others are intended to be subclassed in application-specific pointcuts.

5.2.3.1. Static pointcuts

Static pointcuts are based on method and target class, and cannot take into account the method's arguments. Static pointcuts are sufficient--and best--for most usages. It's possible for Spring to evaluate a static pointcut only once, when a method is first invoked: after that, there is no need to evaluate the pointcut again with each method invocation.

Let's consider some static pointcut implementations included with Spring.

5.2.3.1.1. Regular expression pointcuts

One obvious way to specific static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible. `org.springframework.aop.support.RegexpMethodPointcut` is a generic regular expression pointcut, using Perl 5 regular expression syntax.

Using this class, you can provide a list of pattern Strings. If any of these is a match, the pointcut will evaluate to true. (So the result is effectively the union of these pointcuts.)

The usage is shown below:

```
<bean id="settersAndAbsquatulatePointcut"
    class="org.springframework.aop.support.RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*get.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

A convenience subclass of `RegexpMethodPointcut`, `RegexpMethodPointcutAdvisor`, allows us to reference an Advice also. (Remember that an Advice can be an interceptor, before advice, throws advice etc.) This simplifies wiring, as the one bean serves as both pointcut and advisor, as shown below:

```
<bean id="settersAndAbsquatulateAdvisor"
    class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="interceptor">
        <ref local="beanNameOfAopAllianceInterceptor"/>
    </property>
    <property name="patterns">
        <list>
            <value>.*get.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

RegexpMethodPointcutAdvisor can be used with any Advice type.
The RegexpMethodPointcut class requires the Jakarta ORO regular expression package.

5.2.3.1.2. Attribute-driven pointcuts

An important type of static pointcut is a *metadata-driven* pointcut. This uses the values of metadata attributes: typically, source-level metadata.

5.2.3.2. Dynamic pointcuts

Dynamic pointcuts are costlier to evaluate than static pointcuts. They take into account method *arguments*, as well as static information. This means that they must be evaluated with every method invocation; the result cannot be cached, as arguments will vary.

The main example is the `control flow` pointcut.

5.2.3.2.1. Control flow pointcuts

Spring control flow pointcuts are conceptually similar to AspectJ *cflow* pointcuts, although less powerful. (There is currently no way to specify that a pointcut executes below another pointcut.) A control flow pointcut matches the current call stack. For example, it might fire if the joinpoint was invoked by a method in the `com.mycompany.web` package, or by the `SomeCaller` class. Control flow pointcuts are specified using the `org.springframework.aop.support.ControlFlowPointcut` class.

Note

Control flow pointcuts are significantly more expensive to evaluate at runtime than even other dynamic pointcuts. In Java 1.4, the cost is about 5 times that of other dynamic pointcuts; in Java 1.3 more than 10.

5.2.4. Pointcut superclasses

Spring provides useful pointcut superclasses to help you to implement your own pointcuts.

Because static pointcuts are most useful, you'll probably subclass StaticMethodMatcherPointcut, as shown below. This requires implemented just one abstract method (although it's possible to override other methods to customize behaviour):

```
class TestStaticPointcut extends StaticMethodMatcherPointcut {

    public boolean matches(Method m, Class targetClass) {
        // return true if custom criteria match
    }
}
```

There are also superclasses for dynamic pointcuts.

You can use custom pointcuts with any advice type in Spring 1.0 RC2 and above.

5.2.5. Custom pointcuts

Because pointcuts in Spring are Java classes, rather than language features (as in AspectJ) it's possible to declare custom pointcuts, whether static or dynamic. However, there is no support out of the box for the sophisticated pointcut expressions that can be coded in AspectJ syntax. However, custom pointcuts in Spring can be arbitrarily complex.
Later versions of Spring may offer support for "semantic pointcuts" as offered by JAC: for example, "all methods that change instance variables in the target object."

5.3. Advice types in Spring

Let's now look at how Spring AOP handles advice.

5.3.1. Advice lifecycles

Spring advices can be shared across all advised objects, or unique to each advised object. This corresponds to *per-class* or *per-instance* advice.

Per-class advice is used most often. It is appropriate for generic advice such as transaction advisors. These do not depend on the state of the proxied object or add new state; they merely act on the method and arguments.

Per-instance advice is appropriate for introductions, to support mixins. In this case, the advice adds state to the proxied object.

It's possible to use a mix of shared and per-instance advice in the same AOP proxy.

5.3.2. Advice types in Spring

Spring provides several advice types out of the box, and is extensible to support arbitrary advice types. Let us look at the basic concepts and standard advice types.

5.3.2.1. Interception around advice

The most fundamental advice type in Spring is *interception around advice*.

Spring is compliant with the AOP Alliance interface for around advice using method interception. MethodInterceptors implementing around advice should implement the following interface:

```
public interface MethodInterceptor extends Interceptor {

    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

The *MethodInvocation* argument to the *invoke()* method exposes the method being invoked; the target joinpoint; the AOP proxy; and the arguments to the method. The *invoke()* method should return the invocation's result: the return value of the joinpoint.

A simple *MethodInterceptor* implementation looks as follows:

```
public class DebugInterceptor implements MethodInterceptor {

    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

Note the call to the MethodInvocation's *proceed()* method. This proceeds down the interceptor chain towards the joinpoint. Most interceptors will invoke this method, and return its return value. However, a MethodInterceptor, like any around advice, can return a different value or throw an exception rather than invoke the proceed method. However, you don't want to do this without good reason!
MethodInterceptors offer interoperability with other AOP Alliance-compliant AOP implementations. The other advice types discussed in the remainder of this section implement common AOP concepts, but in a Spring-specific way. While there is an advantage in using the most specific advice type, stick with MethodInterceptor around advice if you are likely to want to run the aspect in another AOP framework. Note that pointcuts are not currently interoperable between frameworks, and the AOP Alliance does not currently define pointcut interfaces.

5.3.2.2. Before advice

A simpler advice type is a **before advice**. This does not need a `MethodInvocation` object, since it will only be called before entering the method.

The main advantage of a before advice is that there is no need to invoke the `proceed()` method, and therefore no possibility of inadvertently failing to proceed down the interceptor chain.

The `MethodBeforeAdvice` interface is shown below. (Spring's API design would allow for field before advice, although the usual objects apply to field interception and it's unlikely that Spring will ever implement it).

```
public interface MethodBeforeAdvice extends BeforeAdvice {

    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

Note the the return type is `void`. Before advice can insert custom behaviour before the joinpoint executes, but cannot change the return value. If a before advice throws an exception, this will abort further execution of the interceptor chain. The exception will propagate back up the interceptor chain. If it is unchecked, or on the signature of the invoked method, it will be passed directly to the client; otherwise it will be wrapped in an unchecked exception by the AOP proxy.

An example of a before advice in Spring, which counts all methods that return normally:

```
public class CountingBeforeAdvice implements MethodBeforeAdvice {
    private int count;
    public void before(Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

Before advice can be used with any pointcut.

5.3.2.3. Throws advice

Throws advice is invoked after the return of the joinpoint if the joinpoint threw an exception. Spring offers typed throws advice. Note that this means that the `org.springframework.aop.ThrowsAdvice` interface does not contain any methods: it is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be of form

```
afterThrowing([Method], [args], [target], subclassOfThrowable)
```

Only the last argument is required. Thus there from one to four arguments, depending on whether the advice method is interested in the method and arguments. The following are examples of throws advices.

This advice will be invoked if a `RemoteException` is thrown (including subclasses):

```
public  class RemoteThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
}
```

The following advice is invoked if a *ServletException* is thrown. Unlike the above advice, it declares 4

arguments, so that it has access to the invoked method, method arguments and target object:

```
public static class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something will all arguments
    }
}
```

The final example illustrates how these two methods could be used in a single class, which handles both `RemoteException` and `ServletException`. Any number of throws advice methods can be combined in a single class.

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }

    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something will all arguments
    }
}
```

Throws advice can be used with any pointcut.

5.3.2.4. After Returning advice

An after returning advice in Spring must implement the *org.springframework.aop.AfterReturningAdvice* interface, shown below:

```
public interface AfterReturningAdvice extends Advice {

    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
            throws Throwable;
}
```

An after returning advice has access to the return value (which it cannot modify), invoked method, methods arguments and target.

The following after returning advice counts all successful method invocations that have not thrown exceptions:

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {
    private int count;

    public void afterReturning(Object returnValue, Method m, Object[] args, Object target) throws Throwable {
        ++count;
    }

    public int getCount() {
        return count;
    }
}
```

This advice doesn't change the execution path. If it throws an exception, this will be thrown up the interceptor chain instead of the return value.
After returning advice can be used with any pointcut.

5.3.2.5. Introduction advice

Spring treats introduction advice as a special kind of interception advice.

Introduction requires an `IntroductionAdvisor`, and an `IntroductionInterceptor`, implementing the

following interface:

```
public interface IntroductionInterceptor extends MethodInterceptor {

    boolean implementsInterface(Class intf);
}
```

The `invoke()` method inherited from the AOP Alliance `MethodInterceptor` interface must implement the introduction: that is, if the invoked method is on an introduced interface, the introduction interceptor is responsible for handling the method call--it cannot invoke `proceed()`.

Introduction advice cannot be used with any pointcut, as it applies only at class, rather than method, level. You can only use introduction advice with the InterceptionIntroductionAdvisor, which has the following methods:

```
public interface InterceptionIntroductionAdvisor extends InterceptionAdvisor {

    ClassFilter getClassFilter();

    IntroductionInterceptor getIntroductionInterceptor();

    Class[] getInterfaces();
}
```

There is no `MethodMatcher`, and hence no `Pointcut`, associated with introduction advice. Only class filtering is logical.

The `getInterfaces()` method returns the interfaces introduced by this advisor.

Let's look at a simple example from the Spring test suite. Let's suppose we want to introduce the following interface to one or more objects:

```
public interface Lockable {
    void lock();
    void unlock();
    boolean locked();
}
```

This illustrates a **mixin**. We want to be able to cast advised objects to Lockable, whatever their type, and call lock and unlock methods. If we call the lock() method, we want all setter methods to throw a `LockedException`. Thus we can add an aspect that provides the ability to make objects immutable, without them having any knowledge of it: a good example of AOP.

Firstly, we'll need an `IntroductionInterceptor` that does the heavy lifting. In this case, we extend the `org.springframework.aop.support.DelegatingIntroductionInterceptor` convenience class. We could implement IntroductionInterceptor directly, but using `DelegatingIntroductionInterceptor` is best for most cases.

The `DelegatingIntroductionInterceptor` is designed to delegate an introduction to an actual implementation of the introduced interface(s), concealing the use of interception to do so. The delegate can be set to any object using a constructor argument; the default delegate (when the no-arg constructor is used) is this. Thus in the example below, the delegate is the `LockMixin` subclass of `DelegatingIntroductionInterceptor`. Given a delegate (by default itself) a `DelegatingIntroductionInterceptor` instance looks for all interfaces implemented by the delegate (other than IntroductionInterceptor), and will support introductions against any of them. It's possible for subclasses such as `LockMixin` to call the `suppressInterflace(Class intf)` method to suppress interfaces that should not be exposed. However, no matter how many interfaces an `IntroductionInterceptor` is prepared to support, the `IntroductionAdvisor` used will control which interfaces are actually exposed. An introduced interface will conceal any implementation of the same interface

by the target.

Thus LockMixin subclasses `DelegatingIntroductionInterceptor` and implements Lockable itself. The superclass automatically picks up that Lockable can be supported for introduction, so we don't need to specify that. We could introduce any number of interfaces in this way.

Note the use of the `locked` instance variable. This effectively adds additional state to that held in the target object.

```
public class LockMixin extends DelegatingIntroductionInterceptor
    implements Lockable {

    private boolean locked;

    public void lock() {
        this.locked = true;
    }

    public void unlock() {
        this.locked = false;
    }

    public boolean locked() {
        return this.locked;
    }

    public Object invoke(MethodInvocation invocation) throws Throwable {
        if (locked() && invocation.getMethod().getName().indexOf("set") == 0)
            throw new LockedException();
        return super.invoke(invocation);
    }

}
```

Often it isn't necessary to override the `invoke()` method: the `DelegatingIntroductionInterceptor` implementation--which calls the delegate method if the method is introduced, otherwise proceeds towards the joinpoint--is usually sufficient. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

The introduction advisor required is simple. All it needs to do is hold a distinct `LockMixin` instance, and specify the introduced interfaces--in this case, just `Lockable`. A more complex example might take a reference to the introduction interceptor (which would be defined as a prototype): in this case, there's no configuration relevant for a `LockMixin`, so we simply create it using `new`.

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

    public LockMixinAdvisor() {
        super(new LockMixin(), Lockable.class);
    }
}
```

We can apply this advisor very simply: it requires no configuration. (However, it *is* necessary: It's impossible to use an `IntroductionInterceptor` without an *IntroductionAdvisor*.) As usual with introductions, the advisor must be per-instance, as it is stateful. We need a different instance of `LockMixinAdvisor`, and hence `LockMixin`, for each advised object. The advisor comprises part of the advised object's state.

We can apply this advisor programmatically, using the `Advised.addAdvisor()` method, or (the recommended way) in XML configuration, like any other advisor. All proxy creation choices discussed below, including "auto proxy creators," correctly handle introductions and stateful mixins.

5.4. Advisors in Spring

In Spring, an Advisor is a modularization of an aspect. Advisors typically incorporate both an advice and a pointcut.

Apart from the special case of introductions, any advisor can be used with any advice. `org.springframework.aop.support.DefaultPointcutAdvisor` is the most commonly used advisor class. For example, it can be used with a `MethodInterceptor`, `BeforeAdvice` or `ThrowsAdvice`.

It is possible to mix advisor and advice types in Spring in the same AOP proxy. For example, you could use a interception around advice, throws advice and before advice in one proxy configuration: Spring will automatically create the necessary create interceptor chain.

5.5. Using the ProxyFactoryBean to create AOP proxies

If you're using the Spring IoC container (an ApplicationContext or BeanFactory) for your business objects--and you should be!--you will want to use one of Spring's AOP FactoryBeans. (Remember that a factory bean introduces a layer of indirection, enabling it to create objects of a different type).

The basic way to create an AOP proxy in Spring is to use the *org.springframework.aop.framework.ProxyFactoryBean*. This gives complete control over the pointcuts and advice that will apply, and their ordering. However, there are simpler options that are preferable if you don't need such control.

5.5.1. Basics

The `ProxyFactoryBean`, like other Spring `FactoryBean` implementations, introduces a level of introduction. If you define a `ProxyFactoryBean` with name `foo`, what objects referencing `foo` see is not the `ProxyFactoryBean` instance itself, but an object created by the `ProxyFactoryBean's` implementation of the `getObject()` method. This method will create an AOP proxy wrapping a target object.

One of the most important benefits of using a `ProxyFactoryBean` or other IoC-aware class to create AOP proxies, is that it means that advices and pointcuts can also be managed by IoC. This is a powerful feature, enabling certain approaches that are hard to achieve with other AOP frameworks. For example, an advice may itself reference application objects (besides the target, which should be available in any AOP framework), benefiting from all the pluggability provided by Dependency Injection.

5.5.2. JavaBean properties

Like most FactoryBean implementations provided with Spring, `ProxyFactoryBean` is itself a JavaBean. Its properties are used to:

- Specify the target you want to proxy
- Specify whether to use CGLIB

Some key properties are inherited from `org.springframework.aop.framework.ProxyConfig`: the superclass for all AOP proxy factories. These include:

- `proxyTargetClass`: true if we should proxy the target class, rather than its interfaces. If this is true we need to use CGLIB.
- `optimize`: whether to apply aggressive optimization to created proxies. Don't use this setting unless you understand how the relevant AOP proxy handles optimization. This is currently used only for CGLIB proxies; it has no effect with JDK dynamic proxies (the default).
- `frozen`: whether advice changes should be disallowed once the proxy factory has been configured. Default is false.
- `exposeProxy`: whether the current proxy should be exposed in a ThreadLocal so that it can be accessed by the target. (It's available via the MethodInvocation without the need for a ThreadLocal.) If a target needs to obtain the proxy and exposeProxy is true, the target can use the `AopContext.currentProxy()` method.
- `aopProxyFactory`: the implementation of AopProxyFactory to use. Offers a way of customizing whether to use dynamic proxies, CGLIB or any other proxy strategy. The default implementation will choose dynamic proxies or CGLIB appropriately. There should be no need to use this property; it's intended to allow the addition of new proxy types in Spring 1.1.

Other properties specific to `ProxyFactory`Bean include:

- `proxyInterfaces`: array of String interface names. If this isn't supplied, a CGLIB proxy for the target class will be used
- `interceptorNames`: String array of Advisor, interceptor or other advice names to apply. Ordering is significant. The names are bean names in the current factory, including bean names from ancestor factories.
- singleton: whether or not the factory should return a single object, no matter how often the `getObject()` method is called. Several `FactoryBean` implementations offer such a method. Default value is true. If you want to use stateful advice--for example, for stateful mixins--use prototype advices along with a singleton value of false.

5.5.3. Proxying interfaces

Let's look at a simple example of ProxyFactoryBean in action. This example involves:

- A target bean that will be proxied. This is the "personTarget" bean definition in the example below.
- An Advisor and an Interceptor used to provide advice.
- An AOP proxy bean definition specifying the target object (the personTarget bean) and the interfaces to proxy, along with the advices to apply.

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name"><value>Tony</value></property>
    <property name="age"><value>51</value></property>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.NopInterceptor">
</bean>

<bean id="person"
    class="org.springframework.aop.framework.ProxyFactoryBean">
```

```
    <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>

    <property name="target"><ref local="personTarget"/></property>
    <property name="interceptorNames">
        <list>
            <value>myAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

Note that the `interceptorNames` property takes a list of String: the bean names of the interceptor or advisors in the current factory. Advisors, interceptors, before, after returning and throws advice objects can be used. The ordering of advisors is significant.
You might be wondering why the list doesn't hold bean references. The reason for this is that if the ProxyFactoryBean's singleton property is set to false, it must be able to return independent proxy instances. If any of the advisors is itself a prototype, and independent instance would need to be returned, so it's necessary to be able to obtain an instance of the prototype from the factory; holding a reference isn't sufficient.

The "person" bean definition above can be used in place of a Person implementation, as follows:

```
Person person = (Person) factory.getBean("person");
```

Other beans in the same IoC context can express a strongly typed dependency on it, as with an ordinary Java object:

```
<bean id="personUser" class="com.mycompany.PersonUser">
    <property name="person"><ref local="person" /></property>
</bean>
```

The `PersonUser` class in this example would expose a property of type Person. As far as it's concerned, the AOP proxy can be used transparently in place of a "real" person implementation. However, its class would be a dynamic proxy class. It would be possible to cast it to the `Advised` interface (discussed below).

5.5.4. Proxying classes

What if you need to proxy a class, rather than one or more interfaces?

Imagine that in our example above, there was no `Person` interface: we needed to advise a class called `Person` that didn't implement any business interface. In this case, you can configure Spring to use CGLIB proxying, rather than dynamic proxies. Simply set the `proxyTargetClass` property on the ProxyFactoryBean above to true. While it's best to program to interfaces, rather than classes, the ability to advise classes that don't implement interfaces can be useful when working with legacy code. (In general, Spring isn't prescriptive. While it makes it easy to apply good practices, it avoids forcing a particular approach.)

If you want to you can force the use of CGLIB in any case, even if you do have interfaces.

CGLIB proxying works by generating a subclass of the target class at runtime. Spring configures this generated subclass to delegate method calls to the original target: the subclass is used to implement the *Decorator* pattern, weaving in the advice.

CGLIB proxying should generally be transparent to users. However, there are some issues to consider:

- `Final` methods can't be advised, as they can't be overridden.
- You'll need the CGLIB 2 binaries on your classpath; dynamic proxies are available with the JDK

There's little performance difference between CGLIB proxying and dynamic proxies. As of Spring 1.0, dynamic proxies are slightly faster. However, this may change in the future. Performance should not be a decisive consideration in this case.

5.6. Convenient proxy creation

Often we don't need the full power of the `ProxyFactoryBean`, because we're only interested in one aspect: For example, transaction management.

There are a number of convenience factories we can use to create AOP proxies when we want to focus on a specific aspect. These are discussed in other chapters, so we'll just provide a quick survey of some of them here.

5.6.1. TransactionProxyFactoryBean

The **jPetStore** sample application shipped with Spring shows the use of the TransactionProxyFactoryBean.

TThe `TransactionProxyFactoryBean` is a subclass of `ProxyConfig`, so basic configuration is shared with `ProxyFactoryBean`. (See list of `ProxyConfig` properties above.)

The following example from the jPetStore illustrates how this works. As with a `ProxyFactoryBean`, there is a target bean definition. Dependencies should be expressed on the proxied factory bean definition ("petStore" here), rather than the target POJO ("petStoreTarget").

The `TransactionProxyFactoryBean` requires a target, and information about "transaction attributes," specifying which methods should be transactional and the required propagation and other settings:

```
<bean id="petStoreTarget" class="org.springframework.samples.jpetstore.domain.logic.PetStoreImpl">
    <property name="accountDao"><ref bean="accountDao"/></property>
    <!-- Other dependencies omitted -->
</bean>

<bean id="petStore"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager"><ref bean="transactionManager"/></property>
    <property name="target"><ref local="petStoreTarget"/></property>
    <property name="transactionAttributes">
        <props>
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

The `TransactionProxyFactoryBean` automatically creates a transaction advisor, including a pointcut based on the transaction attributes, so only transactional methods are advised.

The `TransactionProxyFactoryBean` allows the specification of "pre" and "post" advice, using the preInterceptors and postInterceptors properties. These take Object arrays of interceptors, other advice or Advisors to place in the interception chain before or after the transaction interceptor. These can be populated using a <list> element in XML bean definitions, as follows:

```
<property name="preInterceptors">
    <list>
        <ref local="authorizationInterceptor"/>
        <ref local="notificationBeforeAdvice"/>
    </list>
</property>
<property name="postInterceptors">
    <list>
        <ref local="myAdvisor"/>
    </list>
</property>
```

These properties could be added to the "petStore" bean definition above. A common usage is to combine transactionality with declarative security: a similar approach to that offered by EJB.

Because of the use of actual instance references, rather than bean names as in `ProxyFactoryBean`, pre and post interceptors can be used only for shared-instance advice. Thus they are not useful for stateful advice: for example, in mixins. This is consistent with the TransactionProxyFactoryBean's purpose. It provides a simple way of doing common transaction setup. If you need more complex, customized, AOP, consider using the generic `ProxyFactoryBean`, or an auto proxy creator (see below).
Especially if we view Spring AOP as, in many cases, a replacement for EJB, we find that most advice is fairly generic and uses a shared-instance model. Declarative transaction management and security checks are classic examples.

The `TransactionProxyFactoryBean` depends on a `PlatformTransactionManager` implementation via its `transactionManager` JavaBean property. This allows for pluggable transaction implementation, based on JTA, JDBC or other strategies. This relates to the Spring transaction abstraction, rather than AOP. We'll discuss the transaction infrastructure in the next chapter.
If you're interested only in declarative transaction management, the TransactionProxyFactoryBean is a good solution, and simpler than using a ProxyFactoryBean.

5.6.2. EJB proxies

Other dedicated proxies create proxies for EJBs, enabling the EJB "business methods" interface to be used directly by calling code. Calling code does not need to perform JNDI lookups or use EJB create methods: A significant improvement in readability and architectural flexibility.

See the chapter on Spring EJB services in this manual for further information.

5.7. Creating AOP proxies programmatically with the ProxyFactory

It's easy to create AOP proxies programmatically using Spring. This enables you to use Spring AOP without dependency on Spring IoC.

The following listing shows creation of a proxy for a target object, with one interceptor and one advisor. The interfaces implemented by the target object will automatically be proxied:

```
ProxyFactory factory = new ProxyFactory(myBusinessInterfaceImpl);
factory.addInterceptor(myMethodInterceptor);
factory.addAdvisor(myAdvisor);
MyBusinessInterface tb = (MyBusinessInterface) factory.getProxy();
```

The first step is to contruct a object of type `org.springframework.aop.framework.ProxyFactory`. You can create this with a target object, as in the above example, or specify the interfaces to be proxied in an alternate constructor.

You can add interceptors or advisors, and manipulate them for the life of the ProxyFactory. If you add an IntroductionInterceptionAroundAdvisor you can cause the proxy to implement additional interfaces.

There are also convenience methods on ProxyFactory (inherited from AdvisedSupport) allowing you to add other advice types such as before and throws advice. AdvisedSupport is the superclass of both ProxyFactory and ProxyFactoryBean.
Integrating AOP proxy creation with the IoC framework is best practice in most applications. We recommend that you externalize configuration from Java code with AOP, as in general.

5.8. Manipulating advised objects

However you create AOP proxies, you can manipulate them using the `org.springframework.aop.framework.Advised` interface. Any AOP proxy can be cast to this interface, whatever other interfaces it implements. This interface includes the following methods:

```
void addInterceptor(Interceptor interceptor) throws AopConfigException;

void addInterceptor(int pos, Interceptor interceptor)
        throws AopConfigException;

void addAdvisor(Advisor advisor) throws AopConfigException;

void addAdvisor(int pos, Advisor advisor) throws AopConfigException;

int indexOf(Advisor advisor);

boolean removeAdvisor(Advisor advisor) throws AopConfigException;

void removeAdvisor(int index) throws AopConfigException;

boolean replaceAdvisor(Advisor a, Advisor b) throws AopConfigException;

boolean isFrozen();
```

The `getAdvisors()` method will return an Advisor for every advisor, interceptor or other advice type that has been added to the factory. If you added an Advisor, the returned advisor at this index will be the object that you added. If you added an interceptor or other advice type, Spring will have wrapped this in an advisor with a pointcut that always returns true. Thus if you added a `MethodInterceptor`, the advisor returned for this index will be an `DefaultPointcutAdvisor` returning your `MethodInterceptor` and a pointcut that matches all classes and methods.

The `addAdvisor()` methods can be used to add any Advisor. Usually this will be the generic `DefaultPointcutAdvisor`, which can be used with any advice or pointcut (but not for introduction).

By default, it's possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it's impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)
It's questionable whether it's advisable (no pun intended) to modify advice on a business object in production, although there are no doubt legitimate usage cases. However, it can be very useful in development: for example, in tests. I have sometimes found it very useful to be able to add test code in the form of an interceptor or other advice, getting inside a method invocation I want to test. (For example, the advice can get inside a transaction created for that method: for example, to run SQL to check that a database was correctly updated, before marking the transaction for roll back.)

Depending on how you created the proxy, you can usually set a `frozen` flag, in which case the `Advised` `isFrozen()` method will return true, and any attempts to modify advice through addition or removal will result in an `AopConfigException`. The ability to freeze the state of an advised object is useful in some cases: For example, to prevent calling code removing a security interceptor. It may also be used in Spring 1.1 to allow aggressive optimization if runtime advice modification is known not to be required.

5.9. Using the "autoproxy" facility

So far we've considered explicit creation of AOP proxies using a `ProxyFactoryBean` or similar factory bean.

Spring also allows us to use "autoproxy" bean definitions, which can automatically proxy selected bean definitions. This is built on Spring "bean post processor" infrastructure, which enables modification of any bean definition as the container loads.

In this model, you set up some special bean definitions in your XML bean definition file configuring the auto proxy infrastructure. This allows you just to declare the targets eligible for autoproxying: you don't need to use `ProxyFactoryBean`.

There are two ways to do this:

- Using an autoproxy creator that refers to specific beans in the current context
- A special case of autoproxy creation that deserves to be considered separately; autoproxy creation driven by source-level metadata attributes

5.9.1. Autoproxy bean definitions

The `org.springframework.aop.framework.autoproxy` package provides the following standard autoproxy creators.

5.9.1.1. BeanNameAutoProxyCreator

The BeanNameAutoProxyCreator automatically creates AOP proxies for beans with names matching literal values or wildcards.

```
<bean id="jdkBeanNameProxyCreator"
    class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames"><value>jdk*,onlyJdk</value></property>
    <property name="interceptorNames">
        <list>
            <value>myInterceptor</value>
        </list>
    </property>
</bean>
```

As with `ProxyFactoryBean`, there is an interceptorNames property rather than a list of interceptor, to allow correct behaviour for prototype advisors. Named "interceptors" can be advisors or any advice type.

As with auto proxying in general, the main point of using `BeanNameAutoProxyCreator` is to apply the same configuration consistently to multiple objects, and with minimal volume of configuration. It is a popular choice for applying declarative transactions to multiple objects.

Bean definitions whose names match, such as "jdkMyBean" and "onlyJdk" in the above example, are plain old

bean definitions with the target class. An AOP proxy will be created automatically by the `BeanNameAutoProxyCreator`. The same advice will be applied to all matching beans. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different beans.

5.9.1.2. AdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is `AdvisorAutoProxyCreator`. This will automagically apply eligible advisors in the current context, without the need to include specific bean names in the autoproxy advisor's bean definition. It offers the same merit of consistent configuration and avoidance of duplication as `BeanNameAutoProxyCreator`.

Using this mechanism involves:

- Specifying an `AdvisorAutoProxyCreator` bean definition
- Specifying any number of Advisors in the same or related contexts. Note that these *must* be Advisors, not just interceptors or other advices. This is necessary because there must be a pointcut to evaluate, to check the eligibility of each advice to candidate bean defiitions.

The `AdvisorAutoProxyCreator` will automatically evaluate the pointcut contained in each advisor, to see what (if any) advice it should apply to each business object (such as "businessObject1" and "businessObject2" in the example).

This means that any number of advisors can be applied automatically to each business object. If no pointcut in any of the advisors matches any method in a business object, the object will not be proxied. As bean definitions are added for new business objects, they will automatically be proxied if necessary.

Autoproxying in general has the advantage of making it impossible for callers or dependencies to obtain an un-advised object. Calling getBean("businessObject1") on this ApplicationContext will return an AOP proxy, not the target business object.

```
<bean id="autoProxyCreator"
    class="org.springframework.aop.framework.autoproxy.AdvisorAutoProxyCreator">
</bean>

<bean id="txAdvisor"
    autowire="constructor"
    class="org.springframework.transaction.interceptor.TransactionAttributeSourceTransactionAroundAdvisor">
</bean>

<bean id="customAdvisor"
    class="com.mycompany.MyAdvisor">
</bean>

<bean id="businessObject1"
    class="com.mycompany.BusinessObject1">
    <!-- Properties omitted -->
</bean>

<bean id="businessObject2"
    class="com.mycompany.BusinessObject2">
</bean>
```

The `AdvisorAutoProxyCreator` is very useful if you want to apply the same advice consistently to many business objects. Once the infrastructure definitions are in place, you can simply add new business objects without including specific proxy configuration. You can also drop in additional aspects very easily--for example, tracing or performance monitoring aspects--with minimal change to configuration.

The AdvisorAutoProxyCreator offers support for filtering (using a naming convention so that only certain advisors are evaluated, allowing use of multiple, differently configured, AdvisorAutoProxyCreators in the same factory) and ordering. Advisors can implement the `org.springframework.core.Ordered` interface to ensure correct ordering if this is an issue. The TransactionAttributeSourceTransactionAroundAdvisor used in the above example has a high order value, as most other advice will normally want to be run in a transaction context.

5.9.1.3. AbstractAdvisorAutoProxyCreator

This is the superclass of AdvisorAutoProxyCreator. You can create your own autoproxy creators by subclassing this class, in the unlikely event that advisor definitions offer insufficient customization to the behaviour of the framework `AdvisorAutoProxyCreator`.

5.9.2. Using metadata-driven autoproxying

A particularly important type of autoproxying is driven by metadata. This produces a similar programming model to .NET `ServicedComponents`. Instead of using XML deployment descriptors as in EJB, configuration for transaction management and other enterprise services is held in source-level attributes.

In this case, you use the `AdvisorAutoProxyCreator`, in combination with Advisors that understand metadata attributes. The metadata specifics are held in the pointcut part of the candidate advisors, rather than in the autoproxy creation class itself.

This is really a special case of the `AdvisorAutoProxyCreator`, but deserves consideration on its own. (The metadata-aware code is in the pointcuts contained in the advisors, not the AOP framework itself.)

The `/attributes` directory of the jPetStore sample application shows the use of attribute-driven autoproxying. In this case, there's no need to use the `TransactionProxyFactoryBean`. Simply defining transactional attributes on business objects is sufficient, because of the use of metadata-aware pointcuts. The bean definitions include the following code, in `/WEB-INF/declarativeServices.xml`. Note that this is generic, and can be used outside the jPetStore:

```
<bean id="autoproxy"
    class="org.springframework.aop.framework.autoproxy.AdvisorAutoProxyCreator">
</bean>

<bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource"
    autowire="constructor">
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor"
    autowire="byType">
</bean>

<bean id="transactionAdvisor"
    class="org.springframework.transaction.interceptor.TransactionAttributeSourceTransactionAroundAdvisor"
    autowire="constructor" >
</bean>

<bean id="attributes"
    class="org.springframework.metadata.commons.CommonsAttributes"
/>
```

The `AdvisorAutoProxyCreator` bean definition--called "advisor" in this case, but the name is not significant--will pick up all eligible pointcuts in the current application context. In this case, the "transactionAdvisor" bean definition, of type `TransactionAttributeSourceTransactionAroundAdvisor`, will

apply to classes or methods carrying a transaction attribute. The TransactionAttributeSourceTransactionAroundAdvisor depends on a TransactionInterceptor, via constructor dependency. The example resolves this via autowiring. The `AttributesTransactionAttributeSource` depends on an implementation of the `org.springframework.metadata.Attributes` interface. In this fragement, the "attributes" bean satisfies this, using the Jakarta Commons Attributes API to obtain attribute information. (The application code must have been compiled using the Commons Attributes compilation task.)

The `TransactionInterceptor` defined here depends on a `PlatformTransactionManager` definition, which is not included in this generic file (although it could be) because it will be specific to the application's transaction requirements (typically JTA, as in this example, or Hibernate, JDO or JDBC):

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

If you require only declarative transaction management, using these generic XML definitions will result in Spring automatically proxying all classes or methods with transaction attributes. You won't need to work directly with AOP, and the programming model is similar to that of .NET ServicedComponents.

This mechanism is extensible. It's possible to do autoproxying based on custom attributes. You need to:

- Define your custom attribute.
- Specify an Advisor with the necessary advice, including a pointcut that is triggered by the presence of the custom attribute on a class or method. You may be able to use an existing advice, merely implementing a static pointcut that picks up the custom attribute.

It's possible for such advisors to be unique to each advised class (for example, mixins): they simply need to be defined as prototype, rather than singleton, bean definitions. For example, the `LockMixin` introduction interceptor from the Spring test suite, shown above. could be used in conjunction with an attribute-driven pointcut to target a mixin, as shown here. We use the generic `DefaultPointcutAdvisor`, configured using JavaBean properties:

```
<bean id="lockMixin"
    class="org.springframework.aop.LockMixin"
    singleton="false"
/>

<bean id="lockableAdvisor"
    class="org.springframework.aop.support.DefaultPointcutAdvisor"
    singleton="false"
>
    <property name="pointcut">
        <ref local="myAttributeAwarePointcut"/>
    </property>
    <property name="advice">
        <ref local="lockMixin"/>
    </property>
</bean>

<bean id="anyBean" class="anyclass" ...
```

If the attribute aware pointcut matches any methods in the `anyBean` or other bean definitions, the mixin will be applied. Note that both `lockMixin` and `lockableAdvisor` definitions are prototypes. The `myAttributeAwarePointcut` pointcut can be a singleton definition, as it doesn't hold state for individual advised objects.

5.10. Using TargetSources

Spring offers the concept of a *TargetSource*, expressed in the `org.springframework.aop.TargetSource` interface. This interface is responsible for returning the "target object" implementing the joinpoint. The `TargetSource` implementation is asked for a target instance each time the AOP proxy handles a method invocation.

Developers using Spring AOP don't normally need to work directly with TargetSources, but this provides a powerful means of supporting pooling, hot swappable and other sophisticated targets. For example, a pooling TargetSource can return a different target instance for each invocation, using a pool to manage instances.

If you do not specify a TargetSource, a default implementation is used that wraps a local object. The same target is returned for each invocation (as you would expect).

Let's look at the standard target sources provided with Spring, and how you can use them.
When using a custom target source, your target will usually need to be a prototype rather than a singleton bean definition. This allows Spring to create a new target instance when required.

5.10.1. Hot swappable target sources

The `org.springframework.aop.target.HotSwappableTargetSource` exists to allow the target of an AOP proxy to be switched while allowing callers to keep their references to it.

Changing the target source's target takes effect immediately. The `HotSwappableTargetSource` is threadsafe.

You can change the target via the `swap()` method on HotSwappableTargetSource as follows:

```
HotSwappableTargetSource swapper =
    (HotSwappableTargetSource) beanFactory.getBean("swapper");
Object oldTarget = swapper.swap(newTarget);
```

The XML definitions required look as follows:

```
<bean id="initialTarget" class="mycompany.OldTarget">
</bean>

<bean id="swapper"
    class="org.springframework.aop.target.HotSwappableTargetSource">
    <constructor-arg><ref local="initialTarget"/></constructor-arg>
</bean>

<bean id="swappable"
    class="org.springframework.aop.framework.ProxyFactoryBean"
>
    <property name="targetSource">
        <ref local="swapper"/>
    </property>
</bean>
```

The above `swap()` call changes the target of the swappable bean. Clients who hold a reference to that bean will be unaware of the change, but will immediately start hitting the new target.

Although this example doesn't add any advice--and it's not necessary to add advice to use a `TargetSource`--of course any `TargetSource` can be used in conjunction with arbitrary advice.

5.10.2. Pooling target sources

Using a pooling target source provides a similar programming model to stateless session EJBs, in which a pool

of identical instances is maintained, with method invocations going to free objects in the pool.

A crucial difference between Spring pooling and SLSB pooling is that Spring pooling can be applied to any POJO. As with Spring in general, this service can be applied in a non-invasive way.

Spring provides out-of-the-box support for Jakarta Commons Pool 1.1, which provides a fairly efficient pooling implementation. You'll need the commons-pool Jar on your application's classpath to use this feature. It's also possible to subclass `org.springframework.aop.target.AbstractPoolingTargetSource` to support any other pooling API.

Sample configuration is shown below:

```
<bean id="businessObjectTarget" class="com.mycompany.MyBusinessObject"
    singleton="false">
    ... properties omitted
</bean>

<bean id="poolTargetSource"
    class="org.springframework.aop.target.CommonsPoolTargetSource">
    <property name="targetBeanName"><value>businessObject</value></property>
    <property name="maxSize"><value>25</value></property>
</bean>

<bean id="businessObject"
    class="org.springframework.aop.framework.ProxyFactoryBean"
>
    <property name="targetSource"><ref local="poolTargetSource"/></property>
    <property name="interceptorNames"><value>myInterceptor</value></property>
</bean>
```

Note that the target object--"businessObjectTarget" in the example--*must* be a prototype. This allows the `PoolingTargetSource` implementation to create new instances of the target to grow the pool as necessary. See the Javadoc for `AbstractPoolingTargetSource` and the concrete subclass you wish to use for information about it's properties: maxSize is the most basic, and always guaranteed to be present.

In this case, "myInterceptor" is the name of an interceptor that would need to be defined in the same IoC context. However, it isn't necessary to specify interceptors to use pooling. If you want only pooling, and no other advice, don't set the interceptorNames property at all.

It's possible to configure Spring so as to be able to cast any pooled object to the `org.springframework.aop.target.PoolingConfig` interface, which exposes information about the configuration and current size of the pool through an introduction. You'll need to define an advisor like this:

```
<bean id="poolConfigAdvisor"
    class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
    <property name="target"><ref local="poolTargetSource" /></property>
    <property name="targetMethod"><value>getPoolingConfigMixin</value></property>
</bean>
```

This advisor is obtained by calling a convenience method on the `AbstractPoolingTargetSource` class, hence the use of MethodInvokingFactoryBean. This advisor's name ("poolConfigAdvisor" here) must be in the list of interceptors names in the ProxyFactoryBean exposing the pooled object.

The cast will look as follows:

```
PoolingConfig conf = (PoolingConfig) beanFactory.getBean("businessObject");
System.out.println("Max pool size is " + conf.getMaxSize());
```

Pooling stateless service objects is not usually necessary. We don't believe it should be the default choice, as most stateless objects are naturally threadsafe, and instance pooling is problematic if resources are cached.

Simpler pooling is available using autoproxying. It's possible to set the TargetSources used by any autoproxy creator.

5.10.3. Prototype" target sources

Setting up a "prototype" target source is similar to a pooling TargetSource. In this case, a new instance of the target will be created on every method invocation. Although the cost of creating a new object isn't high in a modern JVM, the cost of wiring up the new object (satisfying its IoC dependencies) may be more expensive. Thus you shouldn't use this approach without very good reason.

To do this, you could modify the `poolTargetSource` definition shown above as follows. (I've also changed the name, for clarity.)

```
<bean id="prototypeTargetSource"
    class="org.springframework.aop.target.PrototypeTargetSource">
    <property name="targetBeanName"><value>businessObject</value></property>
</bean>
```

There's only one property: the name of the target bean. Inheritance is used in the TargetSource implementations to ensure consistent naming. As with the pooling target source, the target bean must be a prototype bean definition.

5.11. Defining new Advice types

Spring AOP is designed to be extensible. While the interception implementation strategy is presently used internally, it is possible to support arbitrary advice types in addition to interception around advice, before, throws advice and after returning advice, which are supported out of the box.

The `org.springframework.aop.framework.adapter` package is an SPI package allowing support for new custom advice types to be added without changing the core framework. The only constraint on a custom Advice type is that it must implement the `org.aopalliance.aop.Advice` tag interface.

Please refer to the `org.springframework.aop.framework.adapter` package's Javadocs for further information

5.12. Further reading and resources

I recommend the excellent *AspectJ in Action* by Ramnivas Laddad (Manning, 2003) for an introduction to AOP.

Please refer to the Spring sample applications for further examples of Spring AOP:

- The JPetStore's default configuration illustrates the use of the TransactionProxyFactoryBean for declarative transaction management
- The `/attributes` directory of the JPetStore illustrates the use of attribute-driven declarative transaction management

If you are interested in more advanced capabilities of Spring AOP, take a look at the test suite. The test

coverage is over 90%, and this illustrates advanced features not discussed in this document.

5.13. Roadmap

Spring AOP, like the rest of Spring, is actively developed. The core API is stable. Like the rest of Spring, the AOP framework is very modular, enabling extension while preserving the fundamental design. Several improvements are likely in the Spring 1.1-1.2 timeframe, which will preserve backward compatibility. These include:

- *Performance improvements*: The creation of AOP proxies is handled by a factory via a Strategy interface. Thus we can support additional AopProxy types without impacting user code or the core implementation. For Spring 1.1, we are examining generating all byte code for AopProxy implementations in cases where runtime advice changes are not required. This should produce a significant reduction in the overhead of the AOP framework. Note, however, that the overhead of the AOP framework is not an issue in normal usage.
- *More expressive pointcuts*: Spring presently offers an expressive Pointcut interface, but we can add value through adding more Pointcut implementations. We are considering implementations offering a simple but powerful expression language. If you wish to contribute a useful Pointcut implementation, please do!
- Introduction of a higher-level concept of an `Aspect`, which could include multiple Advisors.

Chapter 6. Transaction management

6.1. The Spring transaction abstraction

Spring provides a consistent abstraction for transaction management. This abstraction is one of the most important of Spring's abstractiions, and delivers the following benefits:

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, iBATIS Database Layer and JDO.
- Provides a simpler, easier to use, API for programmatic transaction management than most of these transaction APIs
- Integrates with the Spring data access abstraction
- Supports Spring declarative transaction management

Traditionally, J2EE developers have had two choices for transaction management: to use *global* or *local* transactions. Global transactions are managed by the application server, using JTA. Local transactions are resource-specific: for example, a transaction associated with a JDBC connection. This choice had profound implications. Global transactions provide the ability to work with multiple transactional resources. (It's worth noting that most applications use a single transaction resource) With local transactions, the application server is not involved in transaction management, and cannot help ensure correctness across multiple resources.

Global transactions have a significant downside. Code needs to use JTA: a cumbersome API to use (partly due to its exception model). Furthermore, a JTA `UserTransaction` normally needs to be obtained from JNDI: meaning that we need to use *both* JNDI and JTA to use JTA. Obviously all use of global transactions limits the reusability of application code, as JTA is normally only available in an application server environment.

The preferred way to use global transactions was via EJB *CMT* (*Container Managed Transaction*): a form of **declarative transaction management** (as distinguished from **programmatic transaction management**). EJB CMT removes the need for transaction-related JNDI lookups--although of course the use of EJB itself necessitates the use of JNDI. It removes most--not all--need to write Java code to control transactions. The significant downside is that CMT is (obviously) tied to JTA and an application server environment; and that it's only available if we choose to implement business logic in EJBs, or at least behind a transactional EJB facade. The negatives around EJB in general are so great that this is not an attractive proposition, when there are alternatives for declarative transaction management.

Local transactions may be easier to use, but also have significant disadvantages: They cannot work across multiple transactional resources, and tend to invade the programming model. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction.

Spring resolves these problems. It enables application developers to use a consistent programming model *in any environment*. You write your code once, and it can benefit from different transaction management strategies in different environments. Spring provides both declarative and programmatic transaction management. Declarative transaction management is preferred by most users, and recommended in most cases.

With programmatic transaction management developers work with the Spring transaction abstraction, which can run over any underlying transaction infrastructure. With the preferred, declarative, model, developers typically write little or no code related to transaction management, and hence don't depend on Spring's or any other transaction API.

6.2. Transaction strategies

The key to the Spring transaction abstraction is the notion of a *transaction strategy*.

This is captured in the `org.springframework.transaction.PlatformTransactionManager` interface, shown below:

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

    void rollback(TransactionStatus status) throws TransactionException;
}
```

This is primarily an SPI interface, although it can be used programmatically. Note that in keeping with Spring's philosophy, this is an *interface*. Thus it can easily be mocked or stubbed if necessary. Nor is it tied to a lookup strategy such as JNDI: PlatformTransactionManager implementations are defined like any other object in a Spring IoC container. This benefit alone makes this a worthwhile abstraction even when working with JTA: transactional code can be tested much more easily than if it directly used JTA.

In keeping with Spring's philosophy, `TransactionException` is unchecked. Failures of the transaction infrastructure are almost invariably fatal. In rare cases where application code can recover from them, the application developer can still choose to catch and handle `TransactionException`.

The `getTransaction()` method returns a `TransactionStatus` object, depending on a `TransactionDefinition` parameter. The returned `TransactionStatus` might represent a new or existing transaction (if there was a matching transaction in the current call stack).

As with J2EE transaction contexts, a `TransactionStatus` is associated with a **thread** of execution.

The `TransactionDefinition` interface specifies:

- **Transaction isolation**: The degree of isolation this transaction has from the work of other transactions. For example, can this transaction see uncommitted writes from other transactions?
- **Transaction propagation**: Normally all code executed within a transaction scope will run in that transaction. However, there are several options specificying behaviour if a transactional method is executed when a transaction context already exists: For example, simply running in the existing transaction (the commonest case); or suspending the existing transaction and creating a new transaction. Spring offers the transaction propagation options familiar from EJB CMT.
- **Transaction timeout**: How long this transaction may run before timing out (automatically being rolled back by the underlying transaction infrastructure).
- **Read-only status**: A read-only transaction does not modify any data. Read-only transactions can be a useful optimization in some cases (such as when using Hibernate).

These settings reflect standard concepts. If necessary, please refer to a resource discussing transaction isolation levels and other core transaction concepts: Understanding such core concepts is essential to using Spring or any other transaction management solution.

The `TransactionStatus` interface provides a simple way for transactional code to control transaction execution

and query transaction status. The concepts should be familiar, as they are common to all transaction APIs:

```
public interface TransactionStatus {

    boolean isNewTransaction();

    void setRollbackOnly();

    boolean isRollbackOnly();
}
```

However Spring transaction management is used, defining the `PlatformTransactionManager` implementation is essential. In good Spring fashion, this important definition is made using Inversion of Control.

PlatformTransactionManager implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate etc.

The following examples from `dataAccessContext-local.xml` from Spring's **jPetStore** sample application show how a local PlatformTransactionManager implementation can be defined. This will work with JDBC.

We must define a JDBC DataSource, and then use the Spring DataSourceTransactionManager, giving it a reference to the DataSource.

```
<bean id="dataSource"
    class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName"><value>${jdbc.driverClassName}</value></property>
    <property name="url"><value>${jdbc.url}</value></property>
    <property name="username"><value>${jdbc.username}</value></property>
    <property name="password"><value>${jdbc.password}</value></property>
</bean>
```

The PlatformTransactionManager definition will look like this:

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource"><ref local="dataSource"/></property>
</bean>
```

If we use JTA, as in the `dataAccessContext-jta.xml` file from the same sample application, we need to use a container DataSource, obtained via JNDI, and a JtaTransactionManager implementation. The JtaTransactionManager doesn't need to know about the DataSource, or any other specific resources, as it will use the container's global transaction management.

```
<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"><value>jdbc/jpetstore</value></property>
</bean>

<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

We can use Hibernate local transactions easy, as shown in the following examples from the Spring **PetClinic** sample application.

In this case, we need to define a Hibernate LocalSessionFactory, which application code will use to obtain Hibernate Sessions.

The DataSource bean definition will be as either of the above examples, and is not shown. (If it's a container DataSource, it should be non-transactional, as Spring, rather than the container, will manage transactions.)

The "transactionManager" bean in this case is of class HibernateTransactionManager. In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the session factory.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource"><ref local="dataSource"/></property>
    <property name="mappingResources">
        <value>org/springframework/samples/petclinic/hibernate/petclinic.hbm.xml</value>
    </property>
    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">${hibernate.dialect}</prop>
        </props>
    </property>
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.hibernate.HibernateTransactionManager">
    <property name="sessionFactory"><ref local="sessionFactory"/></property>
</bean>
```

With Hibernate and JTA transactions we could simply use the JtaTransactionManager as with JDBC or any other resource strategy.

```
<bean id="transactionManager"
    class="org.springframework.transaction.jta.JtaTransactionManager"/>
```

Note that this is identical to JTA configuration for any resource, as these are global transactions, which can enlist any transactional resource.
In all these cases, application code won't need to change at all. We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

When not using global transactions, you do need to follow one special coding convention. Fortunately this is very simple. You need to obtain connection or session resources in a special way, to allow the relevant PlatformTransactionManager implementation to track connection usage, and apply transaction management as necessary.

For example, if using JDBC, you should not call the `getConnection()` method on a DataSource, but must use the Spring `org.springframework.jdbc.datasource.DataSourceUtils` class as follows:

```
Connection conn = DataSourceUtils.getConnection(dataSource);
```

This has the added advantage that any `SQLException` will be wrapped in a Spring `CannotGetJdbcConnectionException`--one of Spring's hierarchy of unchecked DataAccessExceptions. This gives you more information than can easily be obtained from the `SQLException`, and ensures portability across databases: even across different persistence technologies.

This will work fine without Spring transaction management, so you can use it whether or not you are using Spring for transaction management.

Of course, once you've used Spring's JDBC support or Hibernate support, you won't want to use `DataSourceUtils` or the other helper classes, because you'll be much happier working via the Spring abstraction than directly with the relevant APIs. For example, if you use the Spring JdbcTemplate or jdbc.object package to simplify your use of JDBC, correct connection retrieval happens behind the scenes and you won't need to write any special code.

6.3. Programmatic transaction management

Spring provides two means of programmatic transaction management:

- Using the `TransactionTemplate`
- Using a `PlatformTransactionManager` implementation directly

The first approach is recommended, so we'll focus on it here.

The second approach is similar to using the JTA `UserTransaction` API (although exception handling is less cumbersome).

The `TransactionTemplate` adopts the same approach as other Spring *templates* such as `JdbcTemplate` and `HibernateTemplate`. It uses a callback approach, to free application code from the working of acquiring and releasing resources. (No more try/catch/finally.) Like other templates, a `TransactionTemplate` is threadsafe.

Application code that must execute in a transaction context looks like this. Note that the `TransactionCallback` can be used to return a value:

```
Object result = tt.execute(new TransactionCallback() {
    public Object doInTransaction(TransactionStatus status) {
        updateOperation1();
        return resultOfUpdateOperation2();
    }
});
```

If there's no return value, use a `TransactionCallbackWithoutResult` like this:

```
tt.execute(new TransactionCallbackWithoutResult() {
    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

Code within the callback can role the transaction back by calling the `setRollbackOnly()` method on the `TransactionStatus` object.

Application classes wishing to use the `TransactionTemplate` must have access to a PlatformTransactionManager: usually exposed as a JavaBean property or as a constructor argument.

It's easy to unit test such classes with a mock or stub `PlatformTransactionManager`. There's no JNDI lookup or static magic here: it's a simple interface. As usual, you can use Spring to simplify your unit testing.

6.4. Declarative transaction management

Spring also offers declarative transaction management. This is enabled by Spring AOP.
Most Spring users choose declarative transaction management. It is the option with least impact on application code, and hence is most consistent with the ideals of a non-invasive lightweight container.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with Spring declarative transaction management. The basic approach is similar: It's possible to specify transaction

behaviour (or lack of it) down to individual methods. It's possible to make a `setRollbackOnly()` call within a transaction context if necessary. The differences are:

- Unlike EJB CMT, which is tied to JTA, Spring declarative transaction management works in any environment. It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only.
- Spring enables declarative transaction mangement to be applied to any POJO, not just special classes such as EJBs.
- Spring offers declarative *rollback rules*: a feature with no EJB equivalent, which we'll discuss below. Rollback can be controlled declaratively, not merely programmatically.
- Spring gives you an opportunity to customize transactional behaviour, using AOP. For example, if you want to insert custom behaviour in the case of transaction rollback, you can. You can also add arbitrary advice, along with the transactional advice. With EJB CMT, you have no way to influence the container's transaction management other than `setRollbackOnly()`.
- Spring does not support propagation of transaction contexts across remote calls, as do high-end application servers. If you need this feature, we recommend that you use EJB. However, don't use this feature lightly. Normally we don't want transactions to span remote calls.

The concept of rollback rules is important: they enable us to specify which exceptions (and throwables) should cause automatic roll back. We specify this declaratively, in configuration, not in Java code. So, while we can still call `setRollbackOnly()` on the `TransactionStatus` object to roll the current transaction back programmatically, most often we can specify a rule that `MyApplicationException` should always result in roll back. This has the significant advantage that business objects don't need to depend on the transaction infrastructure. For example, they typically don't need to import any Spring APIs, transaction or other.

While the EJB default behaviour is the for the EJB container to automatically roll back the transaction on a *system exception* (usually a runtime exception), EJB CMT does not roll back the transaction automatically on an *application exception* (checked exception other than `java.rmi.RemoteException`). While the Spring default behaviour for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it's often useful to customize this.

On our benchmarks, the performance of Spring declarative transaction management exceeds that of EJB CMT.

The usual way of setting up transactional proxying in Spring is via the TransactionProxyFactoryBean. We need a target object to wrap in a transactional proxy. The target object is normally a POJO bean definition. When we define the TransactionProxyFactoryBean, we must supply a reference to the relevant PlatformTransactionManager, and **transaction attributes**. Transaction attributes contain the transaction definitions, discussed above.

```
<bean id="petStore"
    class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager"><ref bean="transactionManager"/></property>
    <property name="target"><ref bean="petStoreTarget"/></property>
    <property name="transactionAttributes">
        <props>
            <prop key="insert*">PROPAGATION_REQUIRED,-MyCheckedException</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

The transactional proxy will implement the interfaces of the target: in this case, the bean with id petStoreTarget. (Using CGLIB it's possible to transactionaly proxy a target class. Set the proxyTargetClass property to true for this. It will happen automatically if the target doesn't implement any interfaces. In general, of course, we want to program to interfaces rather than classes.) It's possible (and usually a good idea) to restrict the transactional proxy to proxying only specific target interfaces, using the proxyInterfaces property. It's also possible to customize the behaviour of a TransactionProxyFactoryBean via several properties inherited from `org.springframework.aop.framework.ProxyConfig`, and shared with all AOP proxy factories.

The transactionAttributes here are set using a Properties format defined in the `org.springframework.transaction.interceptor.NameMatchTransactionAttributeSource` class. The mapping from method name, including wildcards, should be fairly intuitive. Note that the value for the insert* mapping contains a rollback rule. Adding `-MyCheckedException` here specifies that if the method throws `MyCheckedException` or any subclasses, the transaction will automatically be rolled back. Multiple rollback rules can be specified here, comma-separated. A - prefix forces rollback; a + prefix specifies commit. (This allows commit even on unchecked exceptions, if you really know what you're doing!)

The `TransactionProxyFactoryBean` allows you to set "pre" and "post" advice, for additional interception behaviour, using the "preInterceptors" and "postInterceptors" properties. Any number of pre and post advices can be set, and their type may be `Advisor` (in which case they can contain a pointcut), `MethodInterceptor` or any advice type supported by the current Spring configuration (such as `ThrowsAdvice`, `AfterReturningtAdvice` or `BeforeAdvice`, which are supported by default.) These advices must support a shared-instance model. If you need transactional proxying with advanced AOP features such as stateful mixins, it's normally best to use the generic `org.springframework.aop.framework.ProxyFactoryBean`, rather than the `TransactionProxyFactoryBean` convenience proxy creator.

It's also possible to set up autoproxying: that is, to configure the AOP framework so that classes are automatically proxied without needing individual proxy definitions.

Please see the chapter on AOP for more information and examples.
You don't need to be an AOP expert--or indeed, to know much at all about AOP--to use Spring's declarative transaction management effectively. However, if you do want to become a "power user" of Spring AOP, you will find it easy to combine declarative transaction management with powerful AOP capabilities.

6.4.1. BeanNameAutoProxyCreator, another declarative approach

`TransactionProxyFactoryBean` is very useful, and gives you full control when wrapping objects with a transactional proxy. In that case that you need to wrap a number of beans in an identical fashion (for example, a boilerplat, 'make all methods transactional', using a `BeanFactoryPostProcessor` called `BeanNameAutoProxyCreator` can offer an alternative approach which can end up being significantly less verbose for this simplified use case.

To recap, once the ApplicaitonContext has read its initialization information, it instantiates any beans within it which implement the `BeanPostProcessor` interface, and gives them a chance to post-process all other beans in the ApplicationContext. So using this mechanism, a properly configured `BeanNameAutoProxyCreator` can be used to postprocess any other beans in the ApplicationContext (recognizing them by name), and wrap them with a transactional proxy. The actual transaction proxy produced is essentially identical to that produced by the use of `TransactionProxyFactoryBean`, so will not be discussed further.

Let us consider a sample configuration:

```
<!-- Transaction Interceptor set up to do PROPOGATION_REQUIRED on all methods -->
<bean id="matchAllWithPropReq"
    class="org.springframework.transaction.interceptor.MatchAlwaysTransactionAttributeSource">
```

```
    <property name="transactionAttribute"><value>PROPAGATION_REQUIRED</value></property>
  </bean>
  <bean id="matchAllTxInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager"><ref bean="transactionManager"/></property>
    <property name="transactionAttributeSource"><ref bean="matchAllWithPropReq"/></property>
  </bean>

  <!-- One BeanNameAutoProxyCreator handles all beans where we want all methods to use
       PROPOGATION_REQUIRED -->
  <bean id="autoProxyCreator"
      class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="interceptorNames">
      <list>T
        <idref local="matchAllTxInterceptor"/>
        <idref bean="hibInterceptor"/>
      </list>
    </property>
    <property name="beanNames">
      <list>
        <idref local="core-services-applicationControllerSevice"/>
        <idref local="core-services-deviceService"/>
        <idref local="core-services-authenticationService"/>
        <idref local="core-services-packagingMessageHandler"/>
        <idref local="core-services-sendEmail"/>
        <idref local="core-services-userService"/>
      </list>
    </property>
  </bean>
```

Assuming that we already have a `TransactionManager` instance in our ApplicationContext, the first thing we need to do is create a `TransactionInterceptor` instance to use. The `TransactionInterceptor` decides what methods to intercept based on a `TransactionAttributeSource` implementing object passed to it as a property. In this case, we want to handle the very simple case of matching all methods. This is not necessarilly the most efficient approach, but it's very quick to set up, because we can use the special pre-defined `MatchAlwaysTransactionAttributeSource`, which simply matches all methods. If we wanted to be more specific, we could use other variants such as `MethodMapTransactionAttributeSource`, `NameMatchTransactionAttributeSource`, or `AttributesTransactionAttributeSource`.

Now that we have the transaction interceptor, we simply feed it to a `BeanNameAutoProxyCreator` instance we define, along with the names of 6 beans in the ApplicationContext that we want to wrap in an identical fashion. As you can see, the net result is significantly less verbose than it would have been to wrap 6 beans identically with TransactionProxyFactoryBean. Wrapping a 7th bean would add only one more line of config.

You may notice that we are able to apply multiple interceptors. In this case, we are also applying a `HibernateInterceptor` we have previously defined (bean id=*hibInterceptor*), which will manage Hibernate Sessions for us.

There is one thing to keep in mind, with regards to bean naming, when switching back and forth between the use of `TransactionProxyFactoryBean`, and `BeanNameAutoProxyCreator`. For the former, you normally give the target bean you want to wrap an id similar in form to *myServiceTarget*, and then give the proxy object an id of *myService*; then all users of the wrapped object simply refer to the proxy, i.e. *myService*. (These are just sample naming conventions, the point is that the target object has a different name than the proxy, and both are available from the ApplicationContext). However, when using `BeanNameAutoProxyCreator`, you name the target object something like *myService*. Then, when `BeanNameAutoProxyCreator` postprocesses the target object and create the proxy, it causes the proxy to be inserted into the Application context under the name of the original bean. From that point on, only the proxy (the wrapped object) is available from the ApplicationContext.

6.5. Choosing between programmatic and declarative transaction management

Programmatic transaction management is usually a good idea only if you have a small number of transactional operations. For example, if you have a web application that require transactions only for certain update operations, you may not want to set up transactional proxies using Spring or any other technology. Using the TransactionTemplate may be a good approach.

On the other hand, if your applications has numerous transactional operations, programmatic transaction management is usually worthwhile. It keeps transaction management out of business logic, and is not difficult to configure in Spring. Using Spring, rather than EJB CMT, the configuration cost of declarative transaction management is greatly reduced.

6.6. Do you need an application server for transaction management?

Spring's transaction management capabilities--and especially its declarative transaction management--significantly changes traditional thinking as to when a J2EE application requires an application server.

In particular, you don't need an application server just to have declarative transactions via EJB. In fact, even if you have an application server with powerful JTA capabilities, you may well decide that Spring declarative transactions offer more power and a much more productive programming model than EJB CMT.

You need an application server's JTA capability only if you need to enlist multiple transactional resources. Many applications don't face this requirement. For example, many high-end applications use a single, highly scalable, database such as Oracle 9i RAC.

Of course you may need other application server capabilities such as JMS and JCA. However, if you need only JTA, you could also consider an open source JTA add-on such as JOTM. (Spring integrates with JOTM out of the box.) However, as of early 2004, high-end application servers provide more robust support for XA transactions.

The most important point is that with Spring *you can choose when to scale your application up to a full-blown application server*. Gone are the days when the only alternative to using EJB CMT or JTA was to write coding using local transactions such as those on JDBC connections, and face a hefty rework if you ever needed that code to run within global, container-managed transactions. With Spring only configuration needs to change: your code doesn't.

6.7. Common problems

Developers should take care to use the correct `PlatformTransactionManager` implementation for their requirements.

It's important to understand how the Spring transaction abstraction works with JTA global transactions. Used properly, there is no conflict here: Spring merely provides a simplifying, portable abstraction.

If you are using global transactions, you *must* use the Spring `org.springframework.transaction.jta.JtaTransactionManager` for all your for all your transactional

operations. Otherwise Spring will attempt to perform local transactions on resources such as container DataSources. Such local transactions don't make sense, and a good application server will treat them as errors.

Chapter 7. Source Level Metadata Support

7.1. Source-level metadata

Source-level metadata is the addition of *attributes* or *annotations* to program elements: usually, classes and/or methods.

For example, we might add metadata to a class as follows:

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.DefaultTransactionAttribute()
 */
public class PetStoreImpl implements PetStoreFacade, OrderService {
```

We could add metadata to a method as follows:

```
/**
 * Normal comments
 * @@org.springframework.transaction.interceptor.RuleBasedTransactionAttribute ()
 * @@org.springframework.transaction.interceptor.RollbackRuleAttribute (Exception.class)
 * @@org.springframework.transaction.interceptor.NoRollbackRuleAttribute ("ServletException")
 */
public void echoException(Exception ex) throws Exception {
    ....
}
```

Both these examples use Jakarta Commons Attributes syntax.

Source-level metadata was introduced to the mainstream with the release of Microsoft's .NET platform, which uses source-level attributes to control transactions, pooling and other behaviour.

The value in this approach has been recognized in the J2EE community. For example, it's much less verbose than the traditional XML deployment descriptors exclusively used by EJB. While it is desirable to externalize some things from program source code, some important enterprise settings--notably transaction characteristics--belong in program source. Contrary to the assumptions of the EJB spec, it seldom makes sense to modify the transactional characteristics of a method.

Although metadata attributes are typically used mainly by framework infrastructure to describe the services application classes require, it should also be possible for metadata attributes to be queried at runtime. This is a key distinction from solutions such as XDoclet, which primarily view metadata as a way of generating code such as EJB artefacts.

There are a number of solutions in this space, including:

- **JSR-175**: the standard Java metadata implementation, available in Java 1.5. But we need a solution now and may always want a facade
- **XDoclet**: well-established solution, primarily intended for code generation
- Various **open source attribute implementations**, pending the release of JSR-175, of which Commons Attributes appears to be the most promising. All these require a special pre- or post-compilation step.

7.2. Spring's metadata support

In keeping with its provision of abstractions over important concepts, Spring provides a facade to metadata implementations, in the form of the `org.springframework.metadata.Attributes` interface.

Such a facade adds value for several reasons:

- There is currently no standard metadata solution. Java 1.5 will provide one, but it is still in beta as of Spring 1.0. Furthermore, there will be a need for metadata support in 1.3 and 1.4 applications for at least two years. Spring aims to provide working solutions *now*; waiting for 1.5 is not an option in such an important area.
- Current metadata APIs, such as Commons Attributes (used by Spring 1.0) are hard to test. Spring provides a simple metadata interface that is much easier to mock.
- Even when Java 1.5 provides metadata support at language level, there will still be value in providing such an abstraction:
    - JSR-175 metadata is static. It is associated with a class at compile time, and cannot be changed in a deployed environment. There is a need for hierarchical metadata, providing the ability to override certain attribute values in deployment--for example, in an XML file.
    - JSR-175 metadata is returned through the Java reflection API. This makes it impossible to mock during test time. Spring provides a simple interface to allow this.

Thus Spring will support JSR-175 before Java 1.5 reaches GA, but will continue to offer an attribute abstraction API.

The Spring `Attributes` interface looks like this:

```
public interface Attributes {

    Collection getAttributes(Class targetClass);

    Collection getAttributes(Class targetClass, Class filter);

    Collection getAttributes(Method targetMethod);

    Collection getAttributes(Method targetMethod, Class filter);

    Collection getAttributes(Field targetField);

    Collection getAttributes(Field targetField, Class filter);
}
```

This is a lowest common denominator interface. JSR-175 offers more capabilities than this, such as attributes on method arguments. As of Spring 1.0, Spring aims to provide the subset of metadata required to provide effective declarative enterprise services a la EJB or .NET. Beyond Spring 1.0, it is likely that Spring will provide further metadata methods.

Note that this interface offers `Object` attributes, like .NET. This distinguishes it from attribute systems such as that of Nanning Aspects and JBoss 4 (as of DR2), which offer only `String` attributes. There is a significant advantage in supporting `Object` attributes. It enables attributes to participate in class hierarchies and enables attributes to react intelligently to their configuration parameters.

In most attribute providers, attribute classes will be configured via constructor arguments or JavaBean

properties. Commons Attributes supports both.

As with all Spring abstraction APIs, `Attributes` is an interface. This makes it easy to mock attribute implementations for unit tests.

7.3. Integration with Jakarta Commons Attributes

Presently Spring supports only Jakarta Commons Attributes out of the box, although it is easy to provide implementations of the `org.springframework.metadata.Attributes` interface for other metadata providers.

Commons Attributes 2.0 (http://jakarta.apache.org/commons/sandbox/attributes/) is a capable attributes solution. It supports attribute configuration via constructor arguments and JavaBean properties, which offers better self-documentation in attribute definitions. (Support for JavaBean properties was added at the request of the Spring team.)

We've already seen two examples of Commons Attributes attributes definitions. In general, we will need to express:

- *The name of the attribute class*. This can be an FQN, as shown above. If the relevant attribute class has already been imported, the FQN isn't required. It's also possible to specify "attribute packages" in attribute compiler configuration.
- *Any necessary parameterization*, via constructor arguments or JavaBean properties

Bean properties look as follows:

```
/**
 * @@MyAttribute(myBooleanJavaBeanProperty=true)
 */
```

It's possible to combine constructor arguments and JavaBean properties (as in Spring IoC).

Because, unlike Java 1.5 attributes, Commons Attributes is not integrated with the Java language, it is necessary to run a special *attribute compilation* step as part of the build process.

To run Commons Attributes as part of the build process, you will need to do the following.

1. Copy the necessary library Jars to `$ANT_HOME/lib`. Four Jars are required, and all are distributed with Spring:

- The Commons Attributes compiler Jar and API Jar
- xjavadoc.jar, from XDoclet
- commons-collections.jar, from Jakarta Commons

2. Import the Commons Attributes ant tasks into your project build script, as follows:

```
<taskdef resource="org/apache/commons/attributes/anttasks.properties"/>
```

3. Next, define an attribute compilation task, which will use the Commons Attributes attribute-compiler task to "compile" the attributes in the source. This process results in the generation of additional sources, to a location specified by the destdir attribute. Here we show the use of a temporary directory:

```
<target name="compileAttributes" >

    <attribute-compiler
        destdir="${commons.attributes.tempdir}"
    >
        <fileset dir="${src.dir}" includes="**/*.java"/>
    </attribute-compiler>

</target>
```

The compile target that runs Javac over the sources should depend on this attribute compilation task, and must also compile the generated sources, which we output to our destination temporary directory. If there are syntax errors in your attribute definitions, they will normally be caught by the attribute compiler. However, if the attribute definitions are syntactically plausible, but specify invalid types or class names, the compilation of the generated attribute classes may fail. In this case, you can look at the generated classes to establish the cause of the problem.
Commons Attributes also provides Maven support. Please refer to Commons Attributes documentation for further information.

While this attribute compilation process may look complex, in fact it's a one-off cost. Once set up, attribute compilation is incremental, so it doesn't usually noticeably slow the build process. And once the compilation process is set up, you may find that use of attributes as described in this chapter can save you a lot of time in other areas.

If you require attribute indexing support (only currently required by Spring for attribute-targeted web controllers, discussed below), you will need an additional step, which must be performed on a Jar file of your compiled classes. In this, optional, step, Commons Attributes will create an index of all the attributes defined on your sources, for efficient lookup at runtime. This step looks as follows:

```
<attribute-indexer jarFile="myCompiledSources.jar">

    <classpath refid="master-classpath"/>

</attribute-indexer>
```

See the /attributes directory of the Spring jPetStore sample application for an example of this build process. You can take the build script it contains and modify it for your own projects.

If your unit tests depend on attributes, try to express the dependency on the Spring Attributes abstraction, rather than Commons Attributes. Not only is this more portable--for example, your tests will still work if you switch to Java 1.5 attributes in future--it simplifies testing. Commons Attributes is a static API, while Spring provides a metadata interface that you can easily mock.

7.4. Metadata and Spring AOP autoproxying

The most important uses of metadata attributes are in conjunction with Spring AOP. This provides a .NET-like programming model, where declarative services are automatically provided to application objects that declare metadata attributes. Such metadata attributes can be supported out of the box by the framework, as in the case of declarative transaction management, or can be custom.

There is widely held to be a synergy between AOP and metadata attributes.

7.4.1. Fundamentals

This builds on the Spring AOP autoproxy functionality. Configuration might look like this:

```
<bean id="autoproxy"
    class="org.springframework.aop.framework.autoproxy.AdvisorAutoProxyCreator">
</bean>

<bean id="transactionAttributeSource"
    class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource"
    autowire="constructor">
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor"
    autowire="byType">
</bean>

<bean id="transactionAdvisor"
    class="org.springframework.transaction.interceptor.TransactionAttributeSourceTransactionAroundAdvisor"
    autowire="constructor" >
</bean>

<bean id="attributes"
    class="org.springframework.metadata.commons.CommonsAttributes"
/>
```

The basic concepts here should be familiar from the discussion of autoproxying in the AOP chapter.

The most important bean definitions are those named **autoproxy** and **transactionAdvisor**. Note that the actual bean names are not important; what matters is their class.

The autoproxy bean definition of class `org.springframework.aop.framework.autoproxy.AdvisorAutoProxyCreator` will automatically advise ("autoproxy") all bean instances in the current factory based on matching Advisor implementations. This class knows nothing about attributes, but relies on Advisors' pointcuts matching. The pointcuts do know about attributes.

Thus we simply need an AOP advisor that will provide declarative transaction management based on attributes.

It's possible to add arbitrary custom Advisor implementations as well, and they will also be evaluated and applied automatically. (You can use Advisors whose pointcuts match on criteria besides attributes in the same autoproxy configuration, if necessary.)

Finally, the `attributes` bean is the Commons Attributes Attributes implementation. Replace with another implementation of `org.springframework.metadata.Attributes` to source attributes from a different source.

7.4.2. Declarative transaction management

The commonest use of source-level attributes it to provide declarative transaction management a la .NET. Once the bean definitions shown above are in place, you can define any number of application objects requiring declarative transactions. Only those classes or methods with transaction attributes will be given transaction advice. You need to do nothing except define the required transaction attributes.

Unlike in .NET, you can specify transaction attributes at either class or method level. Class-level attributes, if specified, will be "inherited" by all methods. Method attributes will wholly override any class-level attributes.

7.4.3. Pooling

Again, as with .NET, you can enable pooling behaviour via class-level attributes. Spring can apply this

behaviour to any POJO. You simply need to specify a pooling attribute, as follows, in the business object to be pooled:

```
 /**
 * @@org.springframework.aop.framework.autoproxy.target.PoolingAttribute (10)
 *
 * @author Rod Johnson
 */
public class MyClass {
```

You'll need the usual autoproxy infrastructure configuration. You then need to specify a pooling `TargetSourceCreator`, as follows. Because pooling affects the creation of the target, we can't use a regular advice. Note that pooling will apply even if there are no advisors applicable to the class, if that class has a pooling attribute.

```
<bean id="poolingTargetSourceCreator"
    class="org.springframework.aop.framework.autoproxy.metadata.AttributesPoolingTargetSourceCreator"
    autowire="constructor" >
</bean>
```

The relevant autoproxy bean definition needs to specify a list of "custom target source creators", including the Pooling target source creator. We could modify the example shown above to include this property as follows:

```
<bean id="autoproxy"
    class="org.springframework.aop.framework.autoproxy.AdvisorAutoProxyCreator">
>
    <property name="customTargetSourceCreators">
        <list>
            <ref local="poolingTargetSourceCreator" />
        </list>
    </property>
</bean>
```

As with the use of metadata in Spring in general, this is a one-off cost: once setup is out of the way, it's very easy to use pooling for additional business objects.
It's arguable that the need for pooling is rare, so there's seldom a need to apply pooling to a large number of business objects. Hence this feature does not appear to be used often.

Please see the Javadoc for the `org.springframework.aop.framework.autoproxy` package for more details. It's possible to use a different pooling implementation than Commons Pool with minimal custom coding.

7.4.4. Custom metadata

We can even go beyond the capabilities of .NET metadata attributes, because of the flexibility of the underlying autoproxying infrastructure.

We can define custom attributes, to provide any kind of declarative behaviour. To do this, you need to:

- Define your custom attribute class
- Define a Spring AOP Advisor with a pointcut that fires on the presence of this custom attribute.
- Add that Advisor as a bean definition to an application context with the generic autoproxy infrastructure in place.

- Add attributes to your POJOs.

There are several potential areas you might want to do this, such as custom declarative security, or possibly caching.
This is a powerful mechanism which can significantly reduce configuration effort in some projects. However, remember that it does rely on AOP under the covers. The more Advisors you have in play, the more complex your runtime configuration will be.
(If you want to see what advice applies to any object, try casting a reference to org.springframework.aop.framework.Advised. This will enable you to examine the Advisors.)

7.5. Using attributes to minimize MVC web tier configuration

The other main use of Spring metadata as of 1.0 is to provide an option to simplify Spring MVC web configuration.

Spring MVC offers flexible *handler mappings*: mappings from incoming request to controller (or other handler) instance. Normally handler mappings are configured in the `xxxx-servlet.xml` file for the relevant Spring DispatcherServlet.

Holding these mappings in the DispatcherServlet configuration file is normally A Good Thing. It provides maximum flexibility. In particular:

- The controller instance is explicitly managed by Spring IoC, through an XML bean definition
- The mapping is external to the controller, so the same controller instance could be given multiple mappings in the same DispatcherServlet context or reused in a different configuration.
- Spring MVC is able to support mappings based on any criteria, rather than merely the request URL-to-controller mappings available in most other frameworks.

However, this does mean that for each controller we typically need both a handler mapping (normally in a handler mapping XML bean definition) and an XML mapping for the controller itself.

Spring offers a simpler approach based on source-level attributes, which is an attractive option in simpler scenarios.
The approach described in this section is best suited to relatively simple MVC scenarios. It sacrifices some of the power of Spring MVC, such as the ability to use the same controller with different mappings, and the ability to base mappings on something other than request URL.

In this approach, controllers are marked with one or more class-level metadata attributes, each specifying one URL they should be mapped to.

The following examples show the approach. In each case, we have a controller that depends on a business object of type Cruncher. As usual, this dependency will be resolved by Dependency Injection. The Cruncher must be available through a bean definition in the relevant DispatcherServlet XML file, or a parent context.

We attach an attribute to the controller class specifying the URL that should map to it. We can express the dependency through a JavaBean property or a constructor argument. This dependency must be resolvable by autowiring: that is, there must be exactly one business object of type Cruncher available in the context.

```
/**
 * Normal comments here
 * @author Rod Johnson
```

```
 * @@org.springframework.web.servlet.handler.metadata.PathMap("/bar.cgi")
 */
public class BarController extends AbstractController {

    private Cruncher cruncher;

    public void setCruncher(Cruncher cruncher) {
        this.cruncher = cruncher;
    }

    protected ModelAndView handleRequestInternal(
            HttpServletRequest arg0, HttpServletResponse arg1)
            throws Exception {
        System.out.println("Bar Crunching c and d =" +
            cruncher.concatenate("c", "d"));
        return new ModelAndView("test");
    }

}
```

For this automapping to work, we need to add the following to the relevant `xxxx-servlet.xml` file, specifying the attributes handler mapping. This special handler mapping can handle any number of controllers with attributes as shown above. The bean id ("commonsAttributesHandlerMapping") is not important. The type is what matters:

```
<bean id="commonsAttributesHandlerMapping"
    class="org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping"
/>
```

We *do not* currently need an Attributes bean definition, as in the above example, because this class works directly with the Commons Attributes API, not via the Spring metadata abstraction.

We now need no XML configuration for each controller. Controllers are automatically mapped to the specified URL(s). Controllers benefit from IoC, using Spring's autowiring capability. For example, the dependency expressed in the "cruncher" bean property of the simple controller shown above is automatically resolved in the current web application context. Both Setter and Constructor Dependency Injection are available, each with zero configuration.

An example of Constructor Injection, also showing multiple URL paths:

```
/**
* Normal comments here
* @author Rod Johnson
*
* @@org.springframework.web.servlet.handler.metadata.PathMap("/foo.cgi")
* @@org.springframework.web.servlet.handler.metadata.PathMap("/baz.cgi")
*/
public class FooController extends AbstractController {

    private Cruncher cruncher;

    public FooController(Cruncher cruncher) {
        this.cruncher = cruncher;
    }

    protected ModelAndView handleRequestInternal(
            HttpServletRequest arg0, HttpServletResponse arg1)
            throws Exception {
        return new ModelAndView("test");
    }

}
```

This approach has the following benefits:

- Significantly reduced volume of configuration. Each time we add a controller we need add *no* XML configuration. As with attribute-driven transaction management, once the basic infrastructure is in place, it is very easy to add more application classes.
- We retain much of the power of Spring IoC to configure controllers.

This approach has the following limitations:

- One-off cost in more complex build process. We need an attribute compilation step and an attribute indexing step. However, once in place, this should not be an issue.
- Currently Commons Attributes only, although support for other attribute providers may be added in future.
- Only "autowiring by type" dependency injection is supported for such controllers. However, this still leaves them far in advance of Struts Actions (with no IoC support from the framework) and, arguably, WebWork Actions (with only rudimentary IoC support) where IoC is concerned.
- Reliance on automagical IoC resolution may be confusing.

Because autowiring by type means there must be exactly one dependency of the specified type, we need to be careful if we use AOP. In the common case using TransactionProxyFactoryBean, for example, we end up with *two* implementations of a business interface such as Cruncher: the original POJO definition, and the transactional AOP proxy. This won't work, as the owning application context can't resolve the type dependency unambiguously. The solution is to use AOP autoproxying, setting up the autoproxy infrastructure so that there is only one implementation of Cruncher defined, and that implementation is automatically advised. Thus this approach works well with attribute-targeted declarative services as described above. As the attributes compilation process must be in place to handle the web controller targeting, this is easy to set up.

Unlike other metadata functionality, there is currently only a Commons Attributes implementation available: org.springframework.web.servlet.handler.metadata.CommonsPathMapHandlerMapping. This limitation is due to the fact that not only do we need attribute compilation, we need attribute *indexing*: the ability to ask the attributes API for all classes with the PathMap attribute. Indexing is not currently offered on the `org.springframework.metadata.Attributes` abstraction interface, although it may be in future. (If you want to add support for another attributes implementation--which must support indexing--you can easily extend the `AbstractPathMapHandlerMapping` superclass of `CommonsPathMapHandlerMapping`, implementing the two protected abstract methods to use your preferred attributes API.)

Thus we need two additional steps in the build process: attribute compilation and attribute indexing. Use of the attribute indexer task was shown above. Note that Commons Attributes presently requires a Jar file as input to indexing.
If you begin with a handler metadata mapping approach, it is possible to switch at any point to a classic Spring XML mapping approach. So you don't close off this option. For this reason, I find that I often start a web application using metadata mapping.

7.6. Other uses of metadata attributes

Other uses of metadata attributes appear to be growing in popularity. As of March 2004, an attribute-based validation package for Spring is in development. The one-off setup cost of attribute parsing looks more attractive, when the potential for multiple uses is considered.

7.7. Adding support for additional metadata APIs

Should you wish to provide support for another metadata API it is easy to do so.

Simply implement the `org.springframework.metadata.Attributes` interface as a facade for your metadata API. You can then include this object in your bean definitions as shown above.

All framework services that use metadata, such as AOP metadata-driven autoproxying, will then automatically be able to use your new metadata provider.
We expect to add support for Java 1.5 attributes--probably as an add-on to the Spring core--in Q2 2004.

Chapter 8. DAO support

8.1. Introduction

The DAO (Data Access Object) support in Spring is primarily aimed at making it easy to work with data access technologies like JDBC, Hibernate or JDO in a standardized way. This allows you to switch between them fairly easily and it also allows you to code without worrying about catching exceptions that are specific to each technology.

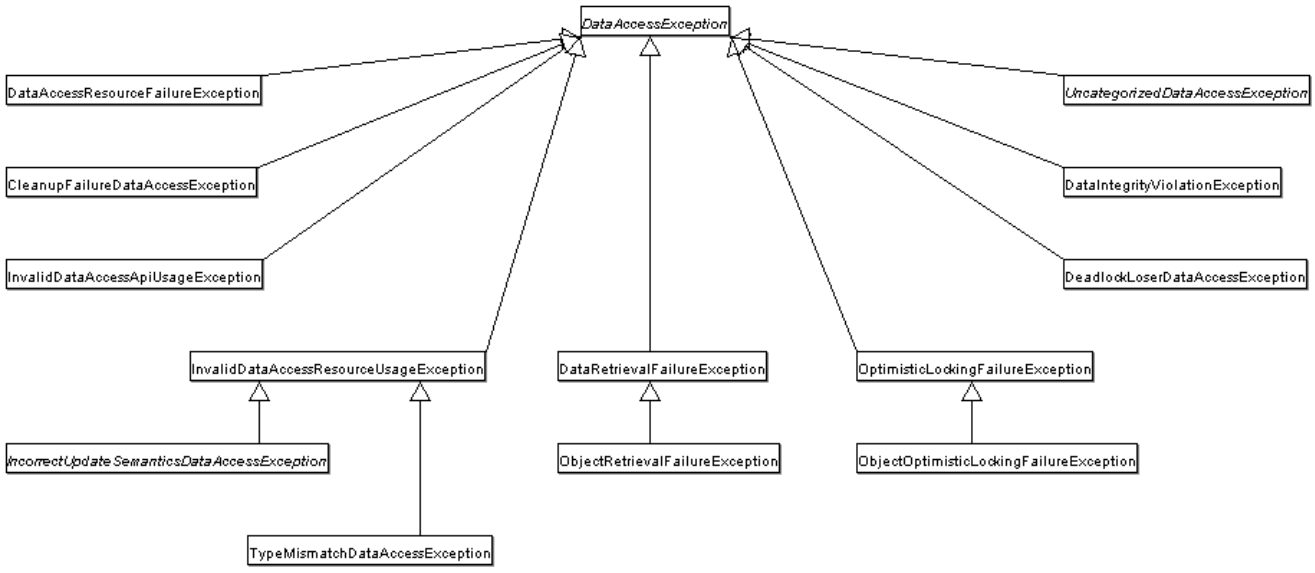
8.2. Consistent Exception Hierarchy

Spring provides a convenient translation from technology specific exceptions like `SQLException` to its own exception hierarchy with the `DataAccessException` as the root exception. These exceptions wrap the original exception so there is never any risk that you would lose any information as to what might have gone wrong.

In addition to JDBC exceptions, Spring can also wrap Hibernate exceptions, converting them from proprietary, checked exceptions, to a set of abstracted runtime exceptions. The same is true for JDO exceptions. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. As we mentioned above, JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

The above is true for the Template versions of the ORM access framework. If you use the Interceptor based classes then the application must care about handling HibernateExceptions and JDOExceptions itself, preferably via delegating to `SessionFactoryUtils`' `convertHibernateAccessException` or `convertJdoAccessException` methods respectively. These methods converts the exceptions to ones that are compatible with the org.springframework.dao exception hierarchy. As JDOExceptions are unchecked, they can simply get thrown too, sacrificing generic DAO abstraction in terms of exceptions though.

The exception hierarchy that Spring uses is outlined in the following graph:



8.3. Consistent Abstract Classes for DAO Support

To make it easier to work with a variety of data access technologies like JDBC, JDO and Hibernate in a consistent way, Spring provides a set of abstract DAO classes that you can extend. These abstract classes has methods for setting the data source and any other configuration settings that are specific to the technology you currently are using.

Dao Support classes:

- `JdbcDaoSupport` - super class for JDBC data access objects. Requires a DataSource to be set, providing a JdbcTemplate based on it to subclasses.
- `HibernateDaoSupport` - super class for Hibernate data access objects. Requires a SessionFactory to be set, providing a HibernateTemplate based on it to subclasses. Can alternatively be initialized directly via a HibernateTemplate, to reuse the latter's settings like SessionFactory, flush mode, exception translator, etc.
- `JdoDaoSupport` - super class for JDO data access objects. Requires a PersistenceManagerFactory to be set, providing a JdoTemplate based on it to subclasses.

Chapter 9. Data Access using JDBC

9.1. Introduction

The JDBC abstraction framework provided by Spring consists of four different packages `core`, `datasource`, `object and support`.

The `org.springframework.jdbc.core` package does as its name suggests contain the classes that provide the core functionality. This includes various SQLExceptionTranslator and DataFieldMaxValueIncrementer implementations as well as a DAO base class for JdbcTemplate usage.

The `org.springframework.jdbc.datasource` package contains a utility class for easy DataSource access, and various simple DataSource implementations that can be used for testing and running unmodified JDBC code outside of a J2EE container. The utility class provides static methods to obtain connections from JNDI and to close connections if necessary. It has support for thread-bound connections, e.g. for use with DataSourceTransactionManager.

Next, the `org.springframework.jdbc.object` package contains classes that represent RDBMS queries, updates, and stored procedures as threadsafe, reusable objects. This approach is modelled by JDO, although of course objects returned by queries are “disconnected” from the database. This higher level of JDBC abstraction depends on the lower-level abstraction in the `org.springframework.jdbc.core` package.

Finally the `org.springframework.jdbc.support` package is where you find the `SQLException` translation functionality and some utility classes.

Exceptions thrown during JDBC processing are translated to exceptions defined in the `org.springframework.dao` package. This means that code using the Spring JDBC abstraction layer does not need to implement JDBC or RDBMS-specific error handling. All translated exceptions are unchecked giving you the option of catching the exceptions that you can recover from while allowing other exceptions to be propagated to the caller.

9.2. Using the JDBC Core classes to control basic JDBC processing and error handling

9.2.1. JdbcTemplate

This is the central class in the JDBC core package. It simplifies the use of JDBC since it handles the creation and release of resources. This helps to avoid common errors like forgetting to always close the connection. It executes the core JDBC workflow like statement creation and execution, leaving application code to provide SQL and extract results. This class executes SQL queries, update statements or stored procedure calls, imitating iteration over ResultSets and extraction of returned parameter values. It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the `org.springframework.dao` package.

Code using this class only need to implement callback interfaces, giving them a clearly defined contract. The `PreparedStatementCreator` callback interface creates a prepared statement given a Connection provided by this class, providing SQL and any necessary parameters. The same is true for the `CallableStatementCreateor` interface which creates callable statement. The `RowCallbackHandler` interface extracts values from each row of a ResultSet.

This class can be used within a service implementation via direct instantiation with a DataSource reference, or get prepared in an application context and given to services as bean reference. Note: The DataSource should always be configured as a bean in the application context, in the first case given to the service directly, in the second case to the prepared template. Because this class is parameterizable by the callback interfaces and the SQLExceptionTranslator interface, it isn't necessary to subclass it. All SQL issued by this class is logged.

9.2.2. DataSource

In order to work with data from a database, we need to obtain a connection to the database. The way Spring does this is through a `DataSource`. A `DataSource` is part of the JDBC specification and can be seen as a generalized connection factory. It allows a container or a framework to hide connection pooling and transaction management issues from the application code. As a developer, you don't need to know any details about how to connect to the database, that is the responsibility for the administrator that sets up the datasource. You will most likely have to fulfil both roles while you are developing and testing you code though, but you will not necessarily have to know how the production data source is configured.

When using Spring's JDBC layer, you can either obtain a data source from JNDI or you can configure your own, using an implementation that is provided in the Spring distribution. The latter comes in handy for unit testing outside of a web container. We will use the `DriverManagerDataSource` implementation for this section but there are several additional implementations that will be covered later on. The `DriverManagerDataSource` works the same way that you probably are used to work when you obtain a JDBC connection. You have to specify the fully qualified class name of the JDBC driver that you are using so that the `DriverManager` can load the driver class. Then you have to provide a url that varies between JDBC drivers. You have to consult the documentation for your driver for the correct value to use here. Finally you must provide a username and a password that will be used to connect to the database. Here is an example of how to configure a `DriverManagerDataSource`:

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
dataSource.setDriverClassName( "org.hsqldb.jdbcDriver");
dataSource.setUrl( "jdbc:hsqldb:hsql://localhost:");
dataSource.setUsername( "sa");
dataSource.setPassword( "");
```

9.2.3. SQLExceptionTranslator

`SQLExceptionTranslator` is an interface to be implemented by classes that can translate between SQLExceptions and our data access strategy-agnostic `org.springframework.dao.DataAccessException`.

Implementations can be generic (for example, using SQLState codes for JDBC) or proprietary (for example, using Oracle error codes) for greater precision.

`SQLErrorCodeSQLExceptionTranslator` is the implementation of SQLExceptionTranslator that is used by default. This implementation uses specific vendor codes. More precise than `SQLState` implementation, but vendor specific. The error code translations are based on codes held in a JavaBean type class named `SQLErrorCodes`. This class is created and populated by an `SQLErrorCodesFactory` which as the name suggests is a factory for creating `SQLErrorCodes` based on the contents of a configuration file named "sql-error-codes.xml". This file is populated with vendor codes and based on the DatabaseProductName taken from the DatabaseMetaData, the codes for the current database are used.

The `SQLErrorCodeSQLExceptionTranslator` applies the following matching rules:

- Try custom translation implemented by any subclass. Note that this class is concrete and is typically used itself, in which case this rule doesn't apply.

- Apply error code matching. Error codes are obtained from the SQLErrorCodesFactory by default. This looks up error codes from the classpath and keys into them from the database name from the database metadata.
- Use the fallback translator. SQLStateSQLExceptionTranslator is the default fallback translator.

`SQLErrorCodeSQLExceptionTranslator` can be extended the following way:

```
public class MySQLErrorCodesTransalator extends SQLErrorCodeSQLExceptionTranslator {
    protected DataAccessException customTranslate(String task, String sql, SQLException sqlex) {
        if (sqlex.getErrorCode() == -12345)
            return new DeadlockLoserDataAccessException(task, sqlex);
        return null;
    }
}
```

In this example the specific error code '-12345' is translated and any other errors are simply left to be translated by the default translator implementation. To use this custom translator, it is necessary to pass it to the `JdbcTemplate` using the method `setExceptionTranslator` and to use this `JdbcTemplate` for all of the data access processing where this translator is needed. Here is an example of how this custom translator can be used:

```
// create a JdbcTemplate and set data source
JdbcTemplate jt = new JdbcTemplate();
jt.setDataSource(dataSource);
// create a custom translator and set the datasource for the default translation lookup
MySQLErrorCodesTransalator tr = new MySQLErrorCodesTransalator();
tr.setDataSource(dataSource);
jt.setExceptionTranslator(tr);
// use the JdbcTemplate for this SqlUpdate
SqlUpdate su = new SqlUpdate();
su.setJdbcTemplate(jt);
su.setSql("update orders set shipping_charge = shipping_charge * 1.05");
su.compile();
su.update();
```

The custom translator is passed a data source because we still want the default translation to look up the error codes in `sql-error-codes.xml`.

9.2.4. Executing Statements

To execute an SQL statement, there is very little code needed. All you need is a `DataSource` and a `JdbcTemplate`. Once you have that, you can use a number of convenience methods that are provided with the `JdbcTemplate`. Here is a short example showing what you need to include for a minimal but fully functional class that creates a new table.

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAStatement {
    private JdbcTemplate jt;
    private DataSource dataSource;

    public void doExecute() {
        jt = new JdbcTemplate(dataSource);
        jt.execute("create table mytable (id integer, name varchar(100))");
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

9.2.5. Running Queries

In addition to the execute methods, there is a large number of query methods. Some of these methods are

intended to be used for queries that return a single value. Maybe you want to retrieve a count or a specific value from one row. If that is the case then you can use `queryForInt`,`queryForLong` or `queryForObject`. The latter will convert the returned JDBC Type to the Java class that is passed in as an argument. If the type conversion is invalid, then an `InvalidDataAccessApiUsageException` will be thrown. Here is an example that contains two query methods, one for an `int` and one that queries for a `String`.

```
import javax.sql.DataSource;
import org.springframework.jdbc.core.JdbcTemplate;

public class RunAQuery {
    private JdbcTemplate jt;
    private DataSource dataSource;

    public int getCount() {
        jt = new JdbcTemplate(dataSource);
        int count = jt.queryForInt("select count(*) from mytable");
        return count;
    }

    public String getName() {
        jt = new JdbcTemplate(dataSource);
        String name = (String) jt.queryForObject("select name from mytable", java.lang.String.class);
        return name;
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

In addition to the singe results query methods there are several methods that return a List with an entry for each row that the query returned. The most generic one is `queryForList` which returns a `List` where each entry is a `Map` with each entry in the map representing the column value for that row. If we add a method to the above example to retrieve a list of all the rows, it would look like this:

```
    public List getList() {
        jt = new JdbcTemplate(dataSource);
        List rows = jt.queryForList("select * from mytable");
        return rows;
    }
```

The list returned would look something like this: `[{name=Bob, id=1}, {name=Mary, id=2}]`.

9.2.6. Updating the database

There are also a number of update methods that you can use. I will show an example where we update a column for a certain primary key. In this example I am using an SQL statement that has place holders for tow parameters. Most of the query and update methods have this functionality. The parameter values are passed in as an array of objects.

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {
    private JdbcTemplate jt;
    private DataSource dataSource;

    public void setName(int id, String name) {
        jt = new JdbcTemplate(dataSource);
        jt.update("update mytable set name = ? where id = ?", new Object[] {name, new Integer(id)});
    }

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
```

```
    }
}
```

9.3. Controling how we connect to the database

9.3.1. DataSourceUtils

Helper class that provides static methods to obtain connections from JNDI and close connections if necessary. Has support for thread-bound connections, e.g. for use with DataSourceTransactionManager.

Note: The getDataSourceFromJndi methods are targeted at applications that do not use a BeanFactory resp. an ApplicationContext. With the latter, it is preferable to preconfigure your beans or even `JdbcTemplate` instances in the factory: `JndiObjectFactoryBean` can be used to fetch a `DataSource` from JNDI and give the `DataSource` bean reference to other beans. Switching to another `DataSource` is just a matter of configuration then: You can even replace the definition of the `FactoryBean` with a non-JNDI `DataSource`!

9.3.2. SmartDataSource

Interface to be implemented by classes that can provide a connection to a relational database. Extends the `javax.sql.DataSource` interface to allow classes using it to query whether or not the connection should be closed after a given operation. This can sometimes be useful for efficiency, if we know that we want to reuse a connection.

9.3.3. AbstractDataSource

Abstract base class for Spring's `DataSource` implementations, taking care of the "uninteresting" glue. This is the class you would extend if you are writing your own `DataSource` implementation.

9.3.4. SingleConnectionDataSource

Implementation of `SmartDataSource` that wraps a single connection which is not closed after use. Obviously, this is not multi-threading capable.

If client code will call close in the assumption of a pooled connection, like when using persistence tools, set `suppressClose` to true. This will return a close-suppressing proxy instead of the physical connection. Be aware that you will not be able to cast this to a native Oracle Connection or the like anymore.

This is primarily a test class. For example, it enables easy testing of code outside an application server, in conjunction with a simple JNDI environment. In contrast to `DriverManagerDataSource`, it reuses the same connection all the time, avoiding excessive creation of physical connections.

9.3.5. DriverManagerDataSource

Implementation of `SmartDataSource` that configures a plain old JDBC Driver via bean properties, and returns a new connection every time.

Useful for test or standalone environments outside of a J2EE container, either as a `DataSource` bean in a respective ApplicationContext, or in conjunction with a simple JNDI environment. Pool-assuming

`Connection.close()` calls will simply close the connection, so any DataSource-aware persistence code should work.

9.3.6. DataSourceTransactionManager

PlatformTransactionManager implementation for single JDBC data sources. Binds a JDBC connection from the specified data source to the thread, potentially allowing for one thread connection per data source.

Application code is required to retrieve the JDBC connection via `DataSourceUtils.getConnection(DataSource)` instead of J2EE's standard `DataSource.getConnection`. This is recommended anyway, as it throws unchecked `org.springframework.dao` exceptions instead of checked `SQLException`. All framework classes like `JdbcTemplate` use this strategy implicitly. If not used with this transaction manager, the lookup strategy behaves exactly like the common one - it can thus be used in any case.

Supports custom isolation levels, and timeouts that get applied as appropriate JDBC statement query timeouts. To support the latter, application code must either use `JdbcTemplate` or call `DataSourceUtils.applyTransactionTimeout` method for each created statement.

This implementation can be used instead of `JtaTransactionManager` in the single resource case, as it does not require the container to support JTA. Switching between both is just a matter of configuration, if you stick to the required connection lookup pattern. Note that JTA does not support custom isolation levels!

9.4. Modeling JDBC operations as Java objects

The `org.springframework.jdbc.object` package contains the classes that allow you to access the database in a more object oriented manner. You can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. You can also execute stored procedures and run update, delete and insert statements.

9.4.1. SqlQuery

Reusable threadsafe object to represent an SQL query. Subclasses must implement the newResultReader() method to provide an object that can save the results while iterating over the ResultSet. This class is rarely used directly since the `MappingSqlQuery`, that extends this class, provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend `SqlQuery` are `MappingSqlQueryWithParameters` and `UpdatableSqlQuery`.

9.4.2. MappingSqlQuery

`MappingSqlQuery` is a reusable query in which concrete subclasses must implement the abstract `mapRow(ResultSet, int)` method to convert each row of the JDBC `ResultSet` into an object.

Of all the `SqlQuery` implementations, this is the one used most often and it is also the one that is the easiest to use.

Here is a brief example of a custom query that maps the data from the customer table to a Java object called Customer.

```
private class CustomerMappingQuery extends MappingSqlQuery {
  public CustomerMappingQuery(DataSource ds) {
    super(ds, "SELECT id, name FROM customer WHERE id = ?");
    super.declareParameter(new SqlParameter("id", Types.INTEGER));
    compile();
```

```
    }
    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
      Customer cust = new Customer();
      cust.setId((Integer) rs.getObject("id"));
      cust.setName(rs.getString("name"));
      return cust;
    }
  }
```

We provide a constructor for this customer query that takes the `DataSource` as the only parameter. In this constructor we call the constructor on the superclass with the `DataSource` and the SQL that should be executed to retrieve the rows for this query. This SQL will be used to create a `PreparedStatement` so it may contain place holders for any parameters to be passed in during execution. Each parameter must be declared using the `declareParameter` method passing in an `SqlParameter`. The `SqlParameter` takes a name and the JDBC type as defined in `java.sql.Types`. After all parameters have been defined we call the `compile` method so the statement can be prepared and later be executed.

Let's take a look at the code where this custom query is instantiated and executed:

```
    public Customer getCustomer(Integer id) {
        CustomerMappingQuery custQry = new CustomerMappingQuery(dataSource);
        Object[] parms = new Object[1];
        parms[0] = id;
        List customers = custQry.execute(parms);
        if (customers.size() > 0)
            return (Customer) customers.get(0);
        else
            return null;
    }
```

The method in this example retrieves the customer with the id that is passed in as the only parameter. After creating an instance of the `CustomerMappingQuery` class we create an array of objects that will contain all parameters that are passed in. In this case there is only one parameter and it is passed in as an `Integer`. Now we are ready to execute the query using this array of parameters and we get a `List` that contains a `Customer` object for each row that was returned for our query. In this case it will only be one entry if there was a match.

9.4.3. SqlUpdate

RdbmsOperation subclass representing a SQL update. Like a query, an update object is reusable. Like all RdbmsOperation objects, an update can have parameters and is defined in SQL.

This class provides a number of update() methods analogous to the execute() methods of query objects.

This class is concrete. Although it can be subclassed (for example to add a custom update method) it can easily be parameterized by setting SQL and declaring parameters.

```
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating  extends SqlUpdate {
    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    /**
```

```
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int run(int id, int rating) {
        Object[] params =
            new Object[] {
                new Integer(rating),
                new Integer(id)};
        return update(params);
    }
}
```

9.4.4. StoredProcedure

Superclass for object abstractions of RDBMS stored procedures. This class is abstract and its execute methods are protected, preventing use other than through a subclass that offers tighter typing.

The inherited sql property is the name of the stored procedure in the RDBMS. Note that JDBC 3.0 introduces named parameters, although the other features provided by this class are still necessary in JDBC 3.0.

Here is an example of a program that calls a function sysdate() that comes with any Oracle database. To use the stored procedure functionality you have to create a class tha extends `StoredProcedure`. There are no input parameters, but there is an output parameter that is declared as a date using the class `SqlOutParameter`. The `execute()` method returns a map with an entry for each declared output parameter using the parameter name as the key.

```
import java.sql.Types;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.datasource.*;
import org.springframework.jdbc.object.StoredProcedure;

public class TestSP {

    public static void main(String[] args)  {

        System.out.println("DB TestSP!");
        TestSP t = new TestSP();
        t.test();
        System.out.println("Done!");

    }

    void test() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.driver.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:mydb");
        ds.setUsername("scott");
        ds.setPassword("tiger");

        MyStoredProcedure sproc = new MyStoredProcedure(ds);
        Map res = sproc.execute();
        printMap(res);

    }

    private class MyStoredProcedure extends StoredProcedure {
        public static final String SQL = "sysdate";

        public MyStoredProcedure(DataSource ds) {
            setDataSource(ds);
            setFunction(true);
            setSql(SQL);
            declareParameter(new SqlOutParameter("date", Types.DATE));
```

```
            compile();
        }

        public Map execute() {
            Map out = execute(new HashMap());
            return out;
        }

    }

    private static void printMap(Map r) {
        Iterator i = r.entrySet().iterator();
        while (i.hasNext()) {
            System.out.println((String) i.next().toString());
        }
    }
}
```

9.4.5. SqlFunction

SQL "function" wrapper for a query that returns a single row of results. The default behavior is to return an int, but that can be overridden by using the methods with an extra return type parameter. This is similar to using the `queryForXxx` methods of the `JdbcTemplate`. The advantage with `SqlFunction` is that you don't have to create the `JdbcTemplate`, it is done behind the scenes.

This class is intended to use to call SQL functions that return a single result using a query like "select user()" or "select sysdate from dual". It is not intended for calling more complex stored functions or for using a `CallableStatement` to invoke a stored procedure or stored function. Use `StoredProcedure` or `SqlCall` for this type of processing.

This is a concrete class, which there is normally no need to subclass. Code using this package can create an object of this type, declaring SQL and parameters, and then invoke the appropriate run method repeatedly to execute the function. Here is an example of retrieving the count of rows from a table:

```
    public int countRows() {
        SqlFunction sf = new SqlFunction(dataSource, "select count(*) from mytable");
        sf.compile();
        return sf.run();
    }
```

Chapter 10. Data Access using O/R Mappers

10.1. Introduction

Spring provides integration with Hibernate, JDO, and iBATIS SQL Maps in terms of resource management, DAO implementation support, and transaction strategies. For Hibernate there is first-class support with lots of IoC convenience features, addressing many typical Hibernate integration issues. All of these comply with Spring's generic transaction and DAO exception hierarchies.

10.2. Hibernate

10.2.1. Overview

Spring adds significant value in a number of areas, when creating data-access applications using Hibernate. You are invited to review and leverage the Spring approach, no matter to what extent, before deciding to take the effort and risk of building similar infrastructure in-house. Much of this Hibernate support code may be used in a library style, as everything is designed as a set of reusable JavaBeans. Usage inside an ApplicationContext or BeanFactory does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside an ApplicationContext.

Benefits of using Spring to create your Hibernate applications include:

- *Session management.* Spring offers efficient, easy, and safe handling of Hibernate Sessions. Related code using Hibernate generally needs to use the same Hibernate Session object for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a session to the current thread, using either a declarative, AOP method interceptor approach, or by using an explicit, *template* wrapper class at the Java code level. Thus Spring solves many of the usage issues that repeatedly arise on the Hibernate forums.

- *General resource management.* Spring application contexts can handle the location and configuration of Hibernate SessionFactories, JDBC datasources, and other related resources. This makes these values easy to manage and change.

- *Integrated transaction management.* Spring allows you to wrap your Hibernate code with either a declarative, AOP style method interceptor, or an explicit 'template' wrapper class at the Java code level. In either case, transaction semantics are handled for you, and proper transaction handling (rollback, etc.) in case of exceptions is taken care of. As discussed below, you also get the benefit of being able to use and swap various transaction managers, without your Hibernate related code being affected. As an added benefit, JDBC-related code can fully integrate transactionally with Hibernate code. This is useful for handling functionality not implemented in Hibernate.

- *Exception wrapping.* Spring can wrap Hibernate exceptions, converting them from proprietary, checked exceptions, to a set of abstracted runtime exceptions. This allows you to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without annoying boilerplate catches/throws, and exception declarations. You can still trap and handle exceptions anywhere you need to. Remember that JDBC exceptions (including DB specific dialects) are also converted to the same hierarchy, meaning that you can perform some operations with JDBC within a consistent programming model.

- *To avoid vendor lock-in, and allow mix-and-match implementation strategies.* While Hibernate is powerful,

flexible, open source and free, it still uses a proprietary API. Given the choice, it's usually desirable to implement major application functionality using standard or abstracted APIs, in case you need to switch to another implementation for reasons of functionality, performance, or any other concerns. Spring's abstraction of Hibernate Transactions and Exceptions, along with its IoC approach which allow you to easily swap in mapper/DAO objects implementing data-access functionality, make it easy to isolate all Hibernate-specific code in one area of your application, without sacrificing any of the power of Hibernate. Higher level service code dealing with the DAOs has no need to know anything about their implementation. This approach has the additional benefit of making it easy to intentionally implement data-access with a mix-and-match approach (i.e. some data-access performed using Hibernate, and some using JDBC) in a non-intrusive fashion, potentially providing great benefits in terms of continuing to use legacy code or leveraging the strength of each technology.

- *Ease of testing.* Spring's inversion of control approach makes it easy to swap the implementations and locations of Hibernate session factories, datasources, transaction managers, and mapper object implementations (if needed). This makes it much easier to isolate and test each piece of persistence-related code in isolation.

10.2.2. Resource Management

Typical business applications are often cluttered with repetitive resource management code. Many projects try to invent their own solutions for this issue, sometimes sacrificing proper handling of failures for programming convenience. Spring advocates strikingly simple solutions for proper resource handling: Inversion of control via templating, i.e. infrastructure classes with callback interfaces, or applying AOP interceptors. The infrastructure cares for proper resource handling, and for appropriate conversion of specific API exceptions to an unchecked infrastructure exception hierarchy. Spring introduces a DAO exception hierarchy, applicable to any data access strategy. For direct JDBC, the `JdbcTemplate` class mentioned in a previous section cares for connection handling, and for proper conversion of `SQLException` to the `DataAccessException` hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. It supports both JTA and JDBC transactions, via respective Spring transaction managers. Spring also offers Hibernate and JDO support, consisting of a `HibernateTemplate` / `JdoTemplate` analogous to `JdbcTemplate`, a `HibernateInterceptor` / `JdoInterceptor`, and a Hibernate / JDO transaction manager. The major goal is to allow for clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business object dependencies on the data access or transaction strategy, no more hard-coded resource lookups, no more hard-to-replace singletons, no more custom service registries. One simple and consistent approach to wiring up application objects, keeping them as reusable and free from container dependencies as possible. All the individual data access features are usable on their own but integrate nicely with Spring's application context concept, providing XML-based configuration and cross-referencing of plain JavaBean instances that don't need to be Spring-aware. In a typical Spring app, many important objects are JavaBeans: data access templates, data access objects (that use the templates), transaction managers, business objects (that use the data access objects and transaction managers), web view resolvers, web controllers (that use the business objects), etc.

10.2.3. Resource Definitions in an Application Context

To avoid tying application objects to hard-coded resource lookups, Spring allows you to define resources like a JDBC DataSource or a Hibernate SessionFactory as beans in an application context. Application objects that need to access resources just receive references to such pre-defined instances via bean references (the DAO definition in the next section illustrates this). The following excerpt from an XML application context definition shows how to set up a JDBC `DataSource` and a Hibernate `SessionFactory` on top of it:

```
<beans>
```

```
    <bean id="myDataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>jdbc/myds</value>
        </property>
    </bean>

   <bean id="mySessionFactory" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
       <property name="mappingResources">
           <list>
               <value>product.hbm.xml</value>
           </list>
       </property>
       <property name="hibernateProperties">
           <props>
               <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
           </props>
       </property>
       <property name="dataSource">
           <ref bean="myDataSource"/>
       </property>
   </bean>

   ...

</beans>
```

Note that switching from a JNDI-located `DataSource` to a locally defined one like a Jakarta Commons DBCP `BasicDataSource` is just a matter of configuration:

```
<bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:hsql://localhost:9001</value>
    </property>
    <property name="username">
        <value>sa</value>
    </property>
    <property name="password">
        <value></value>
    </property>
</bean>
```

You can also use a JNDI-located `SessionFactory`, but that's typically not necessary outside an EJB context (see the "container resources vs local resources" section for a discussion).

10.2.4. Inversion of Control: Template and Callback

The basic programming model for templating looks as follows, for methods that can be part of any custom data access object or business object. There are no restrictions on the implementation of the surrounding object at all, it just needs to provide a Hibernate `SessionFactory`. It can get the latter from anywhere, but preferably as bean reference from a Spring application context - via a simple setSessionFactory bean property setter. The following snippets show a DAO definition in a Spring application context, referencing the above defined `SessionFactory`, and an example for a DAO method implementation.

```
<beans>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    ...

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    public List loadProductsByCategory(final String category) {
        HibernateTemplate hibernateTemplate =
            new HibernateTemplate(this.sessionFactory);

        return (List) hibernateTemplate.execute(
            new HibernateCallback() {
                public Object doInHibernate(Session session) throws HibernateException {
                    List result = session.find(
                        "from test.Product product where product.category=?",
                        category, Hibernate.STRING);
                    // do some further stuff with the result list
                    return result;
                }
            }
        );
    }
}
```

A callback implementation can effectively be used for any Hibernate data access. `HibernateTemplate` will ensure that `Sessions` are properly opened and closed, and automatically participae in transactions. The template instances are thread-safe and reusable, they can thus be kept as instance variables of the surrounding class. For simple single step actions like a single find, load, saveOrUpdate, or delete call, `HibernateTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `HibernateDaoSupport` base class that provides a setSessionFactory method for receiving a SessionFactory, and getSessionFactory and getHibernateTemplate for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public List loadProductsByCategory(String category) {
        return getHibernateTemplate().find(
            "from test.Product product where product.category=?", category,
            Hibernate.STRING);
    }
}
```

10.2.5. Applying an AOP Interceptor Instead of a Template

An alternative to using a `HibernateTemplate` is Spring's AOP `HibernateInterceptor`, replacing the callback implementation with straight Hibernate code within a delegating try/catch block, and a respective interceptor configuration in the application context. The following snippets show respective DAO, interceptor, and proxy definitions in a Spring application context, and an example for a DAO method implementation.

```
<beans>

    ...

    <bean id="myHibernateInterceptor"
        class="org.springframework.orm.hibernate.HibernateInterceptor">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductDaoTarget" class="product.ProductDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>
```

```
    <bean id="myProductDao" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>product.ProductDao</value>
        </property>
        <property name="interceptorNames">
            <list>
                <value>myHibernateInterceptor</value>
                <value>myProductDaoTarget</value>
            </list>
        </property>
    </bean>

    ...

</beans>
```

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public List loadProductsByCategory(final String category) throws MyException {
        Session session = SessionFactoryUtils.getSession(getSessionFactory(), false);
        try {
            List result = session.find(
                "from test.Product product where product.category=?",
                category, Hibernate.STRING);
            if (result == null) {
                throw new MyException("invalid search result");
            }
            return result;
        }
        catch (HibernateException ex) {
            throw SessionFactoryUtils.convertHibernateAccessException(ex);
        }
    }
}
```

This method will only work with a `HibernateInterceptor` for it, caring for opening a thread-bound Session before and closing it after the method call. The "false" flag on getSession makes sure that the Session must already exist; otherwise `SessionFactoryUtils` would create a new one if none was found. If there is already a `SessionHolder` bound to the thread, e.g. by a `HibernateTransactionManager` transaction, `SessionFactoryUtils` automatically takes part in it in any case. `HibernateTemplate` uses `SessionFactoryUtils` internally - it's all the same infrastructure. The major advantage of `HibernateInterceptor` is that it allows any checked application exception to be thrown within the data access code, while HibernateTemplate is restricted to unchecked exceptions within the callback. Note that one can offen defer the respective checks and throwing of application exceptions to after the callback, though. The interceptor's major drawback is that it requires special setup in the context. `HibernateTemplate`'s convenience methods offers simpler means for many scenarios.

10.2.6. Programmatic Transaction Demarcation

On top of such lower-level data access services, transactions can be demarcated in a higher level of the application, spanning any number of operations. There are no restrictions on the implementation of the surrounding business object here too, it just needs a Spring `PlatformTransactionManager`. Again, the latter can come from anywhere, but preferably as bean reference via a `setTransactionManager` method - just like the `productDAO` should be set via a `setProductDao` method. The following snippets show a transaction manager and a business object definition in a Spring application context, and an example for a business method implementation.

```
<beans>

    ...

    <bean id="myTransactionManager"
        class="org.springframework.orm.hibernate.HibernateTransactionManager">
```

```
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductService" class="product.ProductServiceImpl">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
    </bean>

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private PlatformTransactionManager transactionManager;
    private ProductDao productDao;

    public void setTransactionManager(PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        TransactionTemplate transactionTemplate = new TransactionTemplate(this.transactionManager);
        transactionTemplate.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);
        transactionTemplate.execute(
            new TransactionCallbackWithoutResult() {
                public void doInTransactionWithoutResult(TransactionStatus status) {
                    List productsToChange = productDAO.loadProductsByCategory(category);
                    ...
                }
            }
        );
    }
}
```

10.2.7. Declarative Transaction Demarcation

Alternatively, one can use Spring's AOP TransactionInterceptor, replacing the transaction demarcation code with an interceptor configuration in the application context. This allows you to keep business objects free of repetitive transaction demarcation code in each business method. Furthermore, transaction semantics like propagation behavior and isolation level can be changed in a configuration file and do not affect the business object implementations.

```
<beans>

    ...

    <bean id="myTransactionManager"
        class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myTransactionInterceptor"
        class="org.springframework.transaction.interceptor.TransactionInterceptor">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="transactionAttributeSource">
            <value>
```

```
                product.ProductService.increasePrice*=PROPAGATION_REQUIRED
                product.ProductService.someOtherBusinessMethod=PROPAGATION_MANDATORY
            </value>
        </property>
    </bean>

    <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
    </bean>

    <bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <value>product.ProductService</value>
        </property>
        <property name="interceptorNames">
            <list>
                <value>myTransactionInterceptor</value>
                <value>myProductServiceTarget</value>
            </list>
        </property>
    </bean>

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
        this.productDao = productDao;
    }

    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDAO.loadProductsByCategory(category);
        ...
    }

    ...

}
```

As with `HibernateInterceptor`, `TransactionInterceptor` allows any checked application exception to be thrown with the callback code, while `TransactionTemplate` is restricted to unchecked exceptions within the callback. `TransactionTemplate` will trigger a rollback in case of an unchecked application exception, or if the transaction has been marked rollback-only by the application (via `TransactionStatus`). `TransactionInterceptor` behaves the same way by default but allows configurable rollback policies per method. A convenient alternative way of setting up declarative transactions is `TransactionProxyFactoryBean`, particularly if there are no other AOP interceptors involved. `TransactionProxyFactoryBean` combines the proxy definition itself with transaction configuration for a particular target bean. This reduces the configuration effort to one target bean plus one proxy bean. Furthermore, you do not need to specify which interfaces or classes the transactional methods are defined in.

```
<beans>

    ...

    <bean id="myTransactionManager"
        class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name="sessionFactory">
            <ref bean="mySessionFactory"/>
        </property>
    </bean>

    <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
    </bean>
```

```
    <bean id="myProductService"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="target">
            <ref bean="myProductServiceTarget"/>
        </property>
        <property name="transactionAttributes">
            <props>
                <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
                <prop key="someOtherBusinessMethod">PROPAGATION_MANDATORY</prop>
            </props>
        </property>
    </bean>

</beans>
```

10.2.8. Transaction Management Strategies

Both `TransactionTemplate` and `TransactionInterceptor` delegate the actual transaction handling to a `PlatformTransactionManager` instance, which can be a `HibernateTransactionManager` (for a single Hibernate SessionFactory, using a ThreadLocal Session under the hood) or a `JtaTransactionManager` (delegating to the JTA subsystem of the container) for Hibernate applications. You could even use a custom `PlatformTransactionManager` implementation. So switching from native Hibernate transaction management to JTA, i.e. when facing distributed transaction requirements for certain deployments of your application, is just a matter of configuration. Simply replace the Hibernate transaction manager with Spring's JTA transaction implementation. Both transaction demarcation and data access code will work without changes, as they just use the generic transaction management APIs. For distributed transactions across multiple Hibernate session factories, simply combine `JtaTransactionManager` as a transaction strategy with multiple `LocalSessionFactoryBean` definitions. Each of your DAOs then gets one specific SessionFactory reference passed into its respective bean property. If all underlying JDBC data sources are transactional container ones, a business object can demarcate transactions across any number of DAOs and any number of session factories without special regard, as long as it is using `JtaTransactionManager` as the strategy.

```
<beans>

    <bean id="myDataSource1" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>jdbc/myds1</value>
        </property>
    </bean>

    <bean id="myDataSource2" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName">
            <value>jdbc/myds2</value>
        </property>
    </bean>

    <bean id="mySessionFactory1" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="mappingResources">
            <list>
                <value>product.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
            </props>
        </property>
        <property name="dataSource">
            <ref bean="myDataSource1"/>
        </property>
    </bean>

    <bean id="mySessionFactory2" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
```

```
        <property name="mappingResources">
            <list>
                <value>inventory.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect</prop>
            </props>
        </property>
        <property name="dataSource">
            <ref bean="myDataSource2"/>
        </property>
    </bean>

    <bean id="myTransactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager"/>

    <bean id="myProductDao" class="product.ProductDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory1"/>
        </property>
    </bean>

    <bean id="myInventoryDao" class="product.InventoryDaoImpl">
        <property name="sessionFactory">
            <ref bean="mySessionFactory2"/>
        </property>
    </bean>

    <bean id="myProductServiceTarget" class="product.ProductServiceImpl">
        <property name="productDao">
            <ref bean="myProductDao"/>
        </property>
        <property name="inventoryDao">
            <ref bean="myInventoryDao"/>
        </property>
    </bean>

    <bean id="myProductService"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager">
            <ref bean="myTransactionManager"/>
        </property>
        <property name="target">
            <ref bean="myProductServiceTarget"/>
        </property>
        <property name="transactionAttributes">
            <props>
                <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
                <prop key="someOtherBusinessMethod">PROPAGATION_MANDATORY</prop>
            </props>
        </property>
    </bean>

</beans>
```

Both `HibernateTransactionManager` and `JtaTransactionManager` allow for proper JVM-level cache handling with Hibernate - without container-specific transaction manager lookup or JCA connector (as long as not using EJB to initiate transactions). Additionally, `HibernateTransactionManager` can export the JDBC Connection used by Hibernate to plain JDBC access code. This allows for high level transaction demarcation with mixed Hibernate/JDBC data access completely without JTA, as long as just accessing one database!

Note, for an alternative approach to using `TransactionProxyFactoryBean` to declaratively demarcate transactions, please see Section 6.4.1, “BeanNameAutoProxyCreator, another declarative approach”.

10.2.9. Using Spring-managed Application Beans

A Spring application context definition can be loaded with a variety of context implementations, from

`FileSystemXmlApplicationContext` and `ClassPathXmlApplicationContext` to `XmlWebApplicationContext`. This allows for reuse of Spring-managed data access and business objects in all kinds of environments. By default, a web app will have its root context defined in "WEB-INF/applicationContext.xml". In any Spring app, an application context defined in an XML file wires up all the application beans that are involved, from the Hibernate session factory to custom data access and business objects (like the beans above). Most of them do not have to be aware of being managed by the Spring container, not even when collaborating with other beans, as they simply follow JavaBeans conventions. A bean property can either represent a value parameter or a collaborating bean. The following bean definition could be part of a Spring web MVC context that accesses business beans in a root application context.

```
<bean id="myProductList" class="product.ProductListController">
    <property name="productService">
        <ref bean="myProductService"/>
    </property>
</bean>
```

Spring web controllers are provided with all business or data access objects they need via bean references, so there typically isn't any need to do manual bean lookups in the application context. But when using Spring-managed beans with Struts, or within an EJB implementation or even an applet, one is always able to look up a bean manually. Therefore, Spring beans can be leveraged virtually anywhere. One just needs a reference to the application context, be it via a servlet context attribute in the web case, or a manually created instance from a file or class path resource.

```
ApplicationContext context = WebApplicationContextUtils.getWebApplicationContext(servletContext);
ProductService productService = (ProductService) context.getBean("myProductService");
```

```
ApplicationContext context =
    new FileSystemXmlApplicationContext("C:/myContext.xml");
ProductService productService =
    (ProductService) context.getBean("myProductService");
```

```
ApplicationContext context =
    new ClassPathXmlApplicationContext("myContext.xml");
ProductService productService =
    (ProductService) context.getBean("myProductService");
```

10.2.10. Container Resources versus Local Resources

Spring's resource management allows for simple switching between a JNDI SessionFactory and a local one, same for a JNDI DataSource, without having to change a single line of application code. Whether to keep the resource definitions in the container or locally within the application, is mainly a matter of the transaction strategy being used. Compared to a Spring-defined local SessionFactory, a manually registered JNDI SessionFactory does not provide any benefits. If registered via Hibernate's JCA connector, there is the added value of transparently taking part in JTA transactions, especially within EJBs. An important benefit of Spring's transaction support is that it isn't bound to a container at all. Configured to any other strategy than JTA, it will work in a standalone or test environment too. Especially for the typical case of single-database transactions, this is a very lightweight and powerful alternative to JTA. When using local EJB Stateless Session Beans to drive transactions, you depend both on an EJB container and JTA - even if you just access a single database anyway, and just use SLSBs for declarative transactions via CMT. The alternative of using JTA programmatically requires a J2EE environment too. JTA does not just involve container dependencies in terms of JTA itself and of JNDI DataSources. For non-Spring JTA-driven Hibernate transactions, you have to use the Hibernate JCA connector, or extra Hibernate transaction code with JTATransaction being configured, for proper JVM-level caching. Spring-driven transactions can work with a locally defined Hibernate SessionFactory nicely, just like with a local JDBC DataSource - if accessing a single database, of course. Therefore you just have to fall back to

Spring's JTA transaction strategy when actually facing distributed transaction requirements. Note that a JCA connector needs container-specific deployment steps, and obviously JCA support in the first place. This is far more hassle than deploying a simple web app with local resource definitions and Spring-driven transactions. And you often need the Enterprise Edition of your container, as e.g. WebLogic Express does not provide JCA. A Spring app with local resources and transactions spanning one single database will work in any J2EE web container (without JTA, JCA, or EJB) - like Tomcat, Resin, or even plain Jetty. Additionally, such a middle tier can be reused in desktop applications or test suites easily. All things considered: If you do not use EJB, stick with local SessionFactory setup and Spring's `HibernateTransactionManager` or `JtaTransactionManager`. You will get all benefits including proper transactional JVM-level caching and distributed transactions, without any container deployment hassle. JNDI registration of a Hibernate SessionFactory via the JCA connector only adds value for use within EJBs.

10.2.11. Skeletons and Samples

For a commented and detailed sample configuration of a J2EE web app with Spring and Hibernate, have a look at the "webapp-typical" skeleton, included in the Spring Framework distribution. It outlines various data source and transaction manager configuration options that are suitable for JDBC and Hibernate apps. It also shows how to configure AOP interceptors, focussing on the transaction interceptor. As of release 1.0 M2, the Petclinic sample in the Spring distribution offers alternative DAO implementations and application context configurations for JDBC and Hibernate. Petclinic can therefore serve as working sample app that illustrates the use of Hibernate in a Spring web app. It also leverages declarative transaction demarcation with different transaction strategies.

10.3. JDO

ToDO

10.4. iBATIS

ToDo

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, 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.

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. The `DispatcherServlet`

Spring's web framework is - like many other web frameworks - a request driven web framework, designed around a servlet that dispatches requests to controllers 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 allows you to use every other feature Spring has.

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 ???, 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 11.1. Special beans in the WebApplicationContext

Expression	Explanation
handler mapping(s)	(Section 11.4, “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 11.5, “Views and resolving them”) capable of resolving view names and needed by the DispatcherServlet to resolves those views with
locale resolver	(Section 11.6, “Using locales”) capable of resolves the locale a client is using, in order to be able to offer internationalized views
theme resolver	(Section 11.7, “Using themes”) capable of resolving themes your webapplication can use e.g. to offer personalized layouts
multipart resolver	(Section 11.8, “Spring's multipart (fileupload) support”) offers functionality to process file uploads from HTML forms
handlerexception resolver	(???) offers functionality to map exceptions to views or implement other more complex exception handling code

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 affect 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

Exception that might be thrown during processing of the request get picked up by any of the handlerexception resolvers that are declared in the WebApplicationContext. Using those exception resolvers you can define custom behavior in case such exceptions get thrown.

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.

You can customize Spring's DispatcherServlet by adding context parameters in the `web.xml` file or servlet init parameters. The possibilities are listed below.

Table 11.2. DispatcherServlet initialization parameters

Parameter	Explanation
`contextClass`	Class that implements `WebApplicationContext`, which will be used to instantiate the context used by this servlet. If this parameter isn't specified, the `XmlWebApplicationContext` will be used
`contextConfigLocation`	String which is passed to the context instance (specified by `contextClass`) to indicate where context(s) can be found. The String is potentially split up into multiple strings (using a comma as a delimiter) to support multiple contexts (in case of multiple context locations, of beans that are defined twice, the latest takes precedence)
namespace	the namespace of the `WebApplicationContext`. Defaults to `[server-name]-servlet`

11.3. 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.

11.3.1. AbstractController and WebContentGenerator

Of course, jsut a controller interface isn't enough. To provide a basic infrastructure, all Spring's Controller inherit from AbstractController, a class offering caching support and for instance the setting of the mimetype.

Table 11.3. 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

`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("foo", new HashMap());
    }
}
```

```
<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 11.4, “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 11.5, “Views and resolving them” for more information about views).

11.3.2. 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.

11.3.3. 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 11.4. 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

Feature	Explanation
	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>
```

11.3.4. 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

11.4. Handler mappings

Using a handler mapping you can map incoming web requests to appropriate handlers. There are some handler mapping you can use, for example the `SimpleUrlHandlerMapping` or the `BeanNameUrlHandlerMapping`, but

let's first examine the general concept of a `HandlerMapping`.

The functionality a basic `HandlerMapping` provides is the delivering of a `HandlerExecutionChain`, first of all containing one handler that matched the incoming request. The second (but optional) element a handler execution chain will contain is a list of handler interceptor that should be applied to the request. When a request comes in, the `DispatcherServlet` will hand it over to the handler mapping to let it inspect the request and come up with an appropriate HandlerExecutionChain. When done, the DispatcherServlet will execute the handler and interceptors in the chain (if any).

The concept of configurable handler mappings that can optionally contain interceptors (executed before or after the actual handler was executed, or both) is extremely powerful. A lot of supporting functionality can be built-in in custom `HandlerMappings`. Think of a custom handler mapping that chooses a handler not only based on the URL of the request coming in, but also on a specific state of the session associated with the request.

Let's examine the handler mappings that Spring provides.

11.4.1. `BeanNameUrlHandlerMapping`

A very simple, but very powerful handlermapping is the `BeanNameUrlHandlerMapping`, which maps incoming Http-requests to names of beans, defined in the webapplicationcontext. Let's say we want to enable a user to insert an account and we've already provided an appropriate FormController (see Section 11.3.4, “CommandControllers” for more information on Command- and FormControllers) and an JSP view (or Velocity template) that renders the form. When using the BeanNameUrlHandlerMapping, we could map the Http-request with URL `http://samples.com/editaccount.form` to the appropriate FormController as follows:

```
<beans>
    <bean id="handlerMapping"
          class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

    <bean name="/editaccount.form"
          class="org.springframework.web.servlet.mvc.SimpleFormController">
        <property name="formView"><value>account</value></property>
        <property name="successView"><value>account-created</value></property>
        <property name="commandName"><value>Account</value></property>
        <property name="commandClass"><value>samples.Account</value></property>
    </bean>
<beans>
```

All incoming requests for the URL `/editaccount.form` will now be handled by the FormController in the source listing above. Of course we have to define a servlet-mapping in web.xml as well, to let through all the requests ending with .form.

```
<web-app>
    ...
    <servlet>
        <servlet-name>sample</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

   <!-- Maps the sample dispatcher to /*.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

NOTE: if you want to use the `BeanNameUrlHandlerMapping`, you don't necessarily have to define it in the webapplicationcontext (as indicated above). By default, if no handler mapping can be found in the context, the DispatcherServlet creates a `BeanNameUrlHandlerMapping` for you!

11.4.2. `SimpleUrlHandlerMapping`

Another - and much more powerful handlermapping - is the the `SimpleUrlHandlerMapping`. This mapping is configurable in the applicationcontext and has Ant-style pathmatching capabilities (see Section 11.10.1, “A little story about the pathmatcher”). A couple of example will probably makes thing clear enough:

```
<web-app>
    ...
    <servlet>
        <servlet-name>sample</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <!-- Maps the sample dispatcher to /*.form -->
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

Allows all requests ending with .html and .form to be handled by the sample dispatcherservlet.

```
<beans>
    <bean id="handlerMapping"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/*/account.form">editAccountFormController</prop>
                <prop key="/*/editaccount.form">editAccountFormController</prop>
                <prop key="/ex/view*.html">someViewController</prop>
                <prop key="/**/help.html">helpController</prop>
            </props>
        </property>
    </bean>

    <bean id="someViewController"
          class="org.springframework.web.servlet.mvc.FilenameViewController"/>

    <bean id="editAccountFormController"
          class="org.springframework.web.servlet.mvc.SimpleFormController">
        <property name="formView"><value>account</value></property>
        <property name="successView"><value>account-created</value></property>
        <property name="commandName"><value>Account</value></property>
        <property name="commandClass"><value>samples.Account</value></property>
    </bean>
<beans>
```

This handlermapping first of all reroutes all requests in all directories for a file named `help.html` to the `someViewController`, which is a FilenameViewController (more about that can be found in Section 11.3, “Controllers”). Also, all requests for a resource beginning with `view`, ending with `.html`, in the directory `ex`, will be rerouted to that specific controller. Furtermore, two mappings have been defined that will match with the `editAccountFormController`

11.4.3. Adding `HandlerInterceptors`

The handler mapping also has a notion of handler interceptors, that can be extremely useful when you want to apply specific functionality to all requests, for example the checking for a principal or something alike.

Interceptors located in the handler mapping must implement `HandlerInterceptor` from the `org.springframework.web.servlet`-package. This interface defines three methods, one that will be called

before the actual handler will be executed, one that will be called *after* the handler is executed, and one that is called *after the complete request has finished*. Those three methods should provide you with enough flexibility to do all kinds of pre- and postprocessing.

The `preHandle` method has a boolean return value. Using this value, you can tweak the behavior of the execution chain. When returning `true`, the handler execution chain will continue, when returning false, the DispatcherServlet assumes the interceptor itself has taken care of requests (and for instance rendered an appropriate view) and does not continue with executing the other interceptors and the actual handler in the execution chain.

The following example provides an interceptor that intercepts all requests and reroutes the user to a specific page if the time is not between 9 a.m. and 6 p.m.

```
<beans>
    <bean id="handlerMapping"
          class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="officeHoursInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <props>
                <prop key="/*.form">editAccountFormController</prop>
                <prop key="/*.view">editAccountFormController</prop>
            </props>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
          class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime"><value>9</value></property>
        <property name="closingTime"><value>18</value></property>
    </bean>
<beans>
```

```
package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;
    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }
    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }
    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler)
    throws Exception {
        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour < closingTime) {
            return true;
        } else {
            response.sendRedirect("http://host.com/outsideOfficeHours.html");
            return false;
        }
    }
}
```

Any request coming in, will be intercepted by the `TimeBasedAccessInterceptor`, and if the current time is outside office hours, the user will be redirect to a static html file, saying for instance he can only access the website during office hours.

As you can see, Spring has an adapter to make it easy for you to extend the `HandlerInterceptor`.

11.5. Views and resolving them

No MVC framework for web applications without a way to address views. Spring provides view resolvers, which enable you to - without tying yourself to a specific view technology - render models in a browser. Out-of-the-box, Spring enables you to use for example Java Server Pages, Velocity templates and XSLT views. Chapter 12, *Integrating view technologies* has details of integrating various view technologies.

The two classes which are important to the way Spring handles views are the `ViewResolver` and the `View`. The `View` interface addresses the preparation of the request and hands the request over to one of the view technologies. The `ViewResolver` provides a mapping between view names and actual views.

11.5.1. ViewResolvers

As discussed before, all controllers in the SpringWeb framework, return a `ModelAndView` instance. Views in Spring are addressed by a view name and are resolved by a viewresolver. Spring comes with a couple of view resolvers. We'll list most of them as provided by the Spring framework and then provide a couple of examples.

Table 11.5. View resolvers

ViewResolver	Description
AbstractCachingViewResolver	Abstract view resolver taking care of caching views. Lots of views need preparation before they can be used, extending from this viewresolver enables caching of views
ResourceBundleViewResolver	Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle basename. The bundle is typically defined in a properties file, located in the classpath
UrlBasedViewResolver	Simple implementation of ViewResolver that allows for direct resolution of symbolic view names to URLs, without explicit mapping definition. This is appropriate if your symbolic names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings
InternalResourceViewResolver	Convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses like JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass. See UrlBasedViewResolver's javadocs for details
VelocityViewResolver	Convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) and custom subclasses of it

When having JSP for a view technology you can use the for example the `UrlBasedViewResolver`. This view resolver translates view names to a URL and hands the request over the RequestDispatcher to render the view.

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.UrlBasedViewResolver">
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

When returning `test` as a viewname, this view resolver will hand the request over to the RequestDispatcher

that'll send the request to `/WEB-INF/jsp/test.jsp`.

When mixing different view technologies in a webapplications, you can use the ResourceBundleViewResolver:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="baseName"><value>views</value></property>
    <property name="defaultParentView"><value>parentView</value></property
</bean>
```

11.6. Using locales

Most parts of Spring's architecture support internalization, just as the Spring web framework does. SpringWEB enables you to automatically resolve messages using the client's locale. This is done through the so-called `LocaleResolver`, that exists in a couple of useful forms. Defining a localeresolver is pretty simple, just stick it in your webapplication context and you're done. You have to make a decision where to retrieve the locale from however. There are a couple of localeresolver to choose from.

Besides the resolving locales, you can also attach an interceptor to the handlermapping (see Section 11.4.3, “Adding HandlerInterceptors” for more info on that), to change the locale based on for instance a parameter occurring in the request.

When a request comes in, the DispatcherServlet looks for a locale resolver and if it finds one it tries to use it and set the locale. Using the `RequestContext.getLocale()` method, you can always retrieve the locale that was resolved by the locale resolver.

Locale resolvers and interceptors are all defined in the `org.springframework.web.servlet.i18n` package.

11.6.1. `AcceptHeaderLocaleResolver`

This locale resolver inspect the `accept-language` header in the request that was sent by the browser of the client. Usually this header field contains the locale of the client's operating system.

11.6.2. `CookieLocaleResolver`

This locale resolver inspect a Cookie that might exist on the client, to see if there's a locale specified. If so, it uses that specific locale. Using the properties of this locale resolver, you can specify the name of the cookie, as well as the maximum age.

```
<bean id="localeResolver">
    <property name="cookieName"><value>clientlanguage</value></property>
    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
    <property name="cookieMaxAge"><value>100000</value></property>
</bean>
```

This is an example of defining a CookieLocaleResolver.

Table 11.6. Special beans in the WebApplicationContext

Property	Default	Description
cookieName	classname + LOCALE	The name of the cookie
cookieMaxAge	Integer.MAX_INT	The maximum time a cookie will stay persistent on the client. If

Property	Default	Description
		-1 is specified, the cookie will not be persisted, at least, only until the client shuts down his or her browser
cookiePath	/	Using this parameter, you can limit the visibility of the cookie to a certain part of your site. When cookiePath is specified, the cookie will only be visible to that path, and the paths below

11.6.3. `SessionLocaleResolver`

Besides the CookieLocaleResolver and the AcceptHeaderLocaleResolver, you can also use the `SessionLocaleResolver` to retrieve locales from the session that might be associated to the user's request.

11.6.4. `LocaleChangeInterceptor`

You can build in changing of locales using the `LocaleChangeInterceptor`. This interceptor needs to be added to one of the handler mappings (see Section 11.4, “Handler mappings”) and it will detect a parameter in the request and changes the locale (call `setLocale()` on the LocaleResolver that also exists in the context).

```
<bean id="localeChangeInterceptor"
      class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName"><value>siteLanguage</value></property>
</bean>

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref local="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <props>
            <prop key="/**/*.view">someController</prop>
        </props>
    </property>
</bean>
```

All calls to all *.view resources containing a parameter named `siteLanguage` will now change the locale. So a call to `http://www.sf.net/home.view?siteLanguage=nl` will change the site language to Dutch.

11.7. Using themes

Dummy paragraph

11.8. Spring's multipart (fileupload) support

11.8.1. 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.

11.8.2. 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.

11.8.3. 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.

11.9. Handling exceptions

Spring provides so-called `HandlerExceptionResolvers` to ease the pain when unexpected exception might occur while your request is handled by any controller that matched the request. `HandlerExceptionResolvers` somewhat resembles the exception-mappings you can define in the webapplication descriptor `web.xml`. However, they provide a more flexible to handle exceptions. First of all, the HandlerExceptionResolver informs you about what handler was execution while the exception was thrown. Furthermore, a programmatic way of handling exception gives you a lot more options to respond appropriately, then in case the request is forwarded to another URL (which is the case when using the servlet specific exception mappings).

Besides implementing the `HandlerExceptionResolver`, which is only a matter of implementing the `resolveException(Exception, Handler)` method and returning a `ModelAndView`, you can also use the `SimpleMappingExceptionResolver`. This resolver enables you to map the class name of any exception that might be thrown to a view name. With this, you will be do exactly the same the exception mapping feature from the servlet api does, but it's also possible to do more fine grained mappings of exception from different handlers.

11.10. Commonly used utilities

11.10.1. A little story about the pathmatcher

ToDo

Chapter 12. Integrating view technologies

12.1. Introduction

One of the areas in which Spring excels is its separation of view technologies from the rest of the MVC framework. Deciding to use Velocity or XSLT in place of an existing JSP for example, is primarily a matter of configuration. This chapter covers the major view technologies that work with Spring and touches briefly on how to add new ones. This chapter assumes you are already familiar with Section 11.5, “Views and resolving them” which covers the basics of how views in general are coupled to the MVC framework.

12.2. Using Spring together with JSP & JSTL

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views (which in fact happen to be the most popular ones). Using JSP (and also JSTL) is done using a normal viewresolver defined in the WebApplicationContext. Furthermore, of course you need to write some JSPs that will actually render the view. This part describes some of the features and additional things Spring provides to facilitate JSP development.

12.2.1. View resolvers

Just as with any other view technology you're integrating with Spring, for JSPs you'll need a view resolver that will resolve your views. The most commonly used view resolvers when developing with JSPs are the `InternalResourceViewResolver` and the `ResourceBundleViewResolver`. Both are declared in the WebApplicationContext:

```
# The ResourceBundleViewResolver:
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename"><value>views</value></property>
</bean>

# And a sample properties file is uses (views.properties in WEB-INF/classes):
welcome.class=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

productList.class=org.springframework.web.servlet.view.JstlView
productList.url=/WEB-INF/jsp/productlist.jsp
```

As you can see, the ResourceBundleViewResolver needs a properties file defining the view names mapped to 1) a class and 2) a URL. With a ResourceBundleViewResolver you can mix different types of views using only one resolver.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="viewClass"><value>org.springframework.web.servlet.view.JstlView</value></property>
    <property name="prefix"><value>/WEB-INF/jsp/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

The InternalResourceBundleViewResolver can be configured for using JSPs as described above. As probably already stated earlier on, we strongly encourage to place JSP files in a a directory under the WEB-INF directory, since there they can't be access directly by clients.

12.2.2. 'Plain-old' JSPs versus JSTL

When using Java Standard Tag Library you need to use a special viewclass, i.e. the `JstlView`. JSTL needs some preparation before things like the i18N features. So remember: when using JSTL, use the JstlView.

12.2.3. Additional tags facilitating development

Spring provides data binding of request parameters to command objects as described in earlier chapters. To facilitate the development of JSP pages in combination with those data binding features, Spring provides a couple of tags that ease the pain (if that should even arise ;-). All Spring have *html escaping* features to enable or disable escaping of characters.

The tag libarry descriptor (TLD) is included in the `spring.jar` as well in the distribution itself. More information about the individual tags can be found online: http://www.springframework.org/docs/taglib/index.html.

12.3. Using Tiles

It is possible to integrate Tiles - just as any other view technology - in webapplications using Spring. The following describes in a broad way how to do this.

12.3.1. Dependencies

To be able to use Tiles you have to have a couple of additional dependencies including in your project. The following is the list of dependencies you need.

- `struts version 1.1`
- `commons-beanutils`
- `commons-digester`
- `commons-logging`
- `commons-lang`

The dependencies are all available in the Spring distribution.

12.3.2. How to integrate Tiles

To be able to use Tiles, you have to configure it using files containing definitions (for basic information on definitions and other Tiles concepts, please have a look at http://jakarta.apache.org/struts. In Spring this is done using the `TilesConfigurer`. Have a look at the following piece of code (which you should include in your WebApplicationContext):

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
    <property name="factoryClass">
        <value>org.apache.struts.tiles.xmlDefinition.I18nFactorySet</value>
    </property>
    <property name="definitions">
        <list>
            <value>/WEB-INF/defs/general.xml</value>
            <value>/WEB-INF/defs/widgets.xml</value>
            <value>/WEB-INF/defs/administrator.xml</value>
            <value>/WEB-INF/defs/customer.xml</value>
            <value>/WEB-INF/defs/templates.xml</value>
        </list>
    </property>
</bean>
```

As you can see, there are five files containing definitions, which are all located in the WEB-INF/defs directory. At initialization of the WebApplicationContext, the files will be loaded and the definitionsfactory defined by the `factoryClass`-property is initialized. After that has been done, the tiles includes in the definition files can be used as views within your Spring webapplication. To be able to use the views you have to have a `ViewResolver` just as with any other view technology used with Spring. Below you can find two possibilities, the `InternalResourceViewResolver` and the `ResourceBundleViewResolver`.

12.3.2.1. `InternalResourceViewResolver`

The InternalResourceViewResolver instantiates the given `viewClass` for each view it has to resolve.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="requestContextAttribute"><value>requestContext</value></property>
    <property name="viewClass">
        <value>org.springframework.web.servlet.view.tiles.TilesView</value>
    </property>
</bean>
```

12.3.2.2. `ResourceBundleViewResolver`

The ResourceBundleViewResolver has to be provided with a property file containing viewnames and viewclasses the resolver can use:

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename"><value>views</value></property>
</bean>
```

```
    ...
    welcomeView.class=org.springframework.web.servlet.view.tiles.TilesView
    welcomeView.url=welcome (<b>this is the name of a definition</b>)

    vetsView.class=org.springframework.web.servlet.view.tiles.TilesView
    vetsView.url=vetsView (<b>again, this is the name of a definition</b>)

    findOwnersForm.class=org.springframework.web.servlet.view.JstlView
    findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp
    ...
```

As you can see, when using the ResourceBundleViewResolver, you can mix view using different view technologies.

12.4. Velocity

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

12.4.1. 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.jar in your `WEB-INF/lib` folder too! The latest stable velocity jar is supplied with the Spring framework and can be copied from the `/lib/velocity` directory.

12.4.2. 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>
```

12.4.3. 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.

12.4.3.1. Template Locations

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`

12.4.3.2. Example velocity.properties

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 you 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/5.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 (i.e. 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
```

12.4.3.3. Web application root marker

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.

12.4.3.4. Alternative property specifications

Alternatively, you can specify velocity properties directly in the bean definition for the Velocity config bean by replacing the "configLocation" property with the following inline properties..

```
<property name="velocityProperties">
    <props>
        <prop key="resource.loader">file</prop>
        <prop key="file.resource.loader.class">org.apache.velocity.runtime.resource.loader.FileResourceLoader</p
        <prop key="file.resource.loader.path">${webapp.root}/WEB-INF/velocity</prop>
        <prop key="file.resource.loader.cache">false</prop>
    </props>
</property>
```

12.4.3.5. Default configuration (file resource loaders)

Note that as of Spring version 1.0-m4 you can avoid using a properties file or inline properties to define file

system loading of templates by putting the following property in the Velocity config bean which will cause the other values to be figured out.

```
<property name="resourceLoaderPath"><value>/WEB-INF/velocity/</value></property>
```

12.4.4. 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 the 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="url"><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="url"><value>secondaryTemplate.vm</value></property>
    <property name="attributes">
        <props>
            <prop key="title">My Velocity Secondary Page</prop>
        </props>
    </property>
</bean>
```

12.4.5. Creating the Velocity templates

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>

        ## model objects are set in the controller and referenced
        ## via bean properties o method names.  See the Velocity
        ## docs for info

        Model Value: $model.value
        Model Method Return Value: $model.getReturnVal()
```

```
    </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.

12.4.6. Form Handling

Spring provides a tag library for use in JSP's that contains (amongst other things) a `<spring:bind>` tag. This tag primarily enables forms to display the results of failed validations from a `Validator` in the web or business tier. This behaviour can be simulated with a Velocity macro and some additional Spring functionality.

12.4.6.1. Validation errors

The error messages that are actually produced from the validation of a form submission can be read from a properties file to make them easily maintainable or internationalised. Spring handles this elegantly in its own way and you should refer to the MVC tutorial or relevant parts of the javadocs for details on how this works. In order to gain access to the messages, the `RequestContext` object needs to be exposed to your Velocity templates in the `VelocityContext`. Amend your template definition in views.properties or views.xml file to give a name to this attribute (giving it a name is what causes it to be exposed)

```
<bean id="welcomeView" parent="parentVelocityView">
    <property name="requestContextAttribute"><value>rc</value></property>
    <property name="attributes">
        <props>
            <prop key="title">My Velocity Home Page</prop>
        </props>
    </property>
</bean>
```

The example above is based on our earlier example and sets the RequestContext attribute name to be rc. All Velocity views that inherit from this parent view will now have an additional object available to them via the reference `$rc`

12.4.6.2. Velocity macro helper

Next, a velocity macro needs to be defined. It makes sense to create this macro in a global macro file since it will be reusable across many Velocity templates (html forms). Refer to the Velocity documentation for more information on creating macros.

The code below should go into the file VM_global_library.vm in the root directory of your Velocity template location..

```
#*
 * showerror
 *
 * display an error for the field name supplied if one exists
 * in the supplied errors object.
 *
 * param $errors the object obtained from RequestContext.getErrors( "formBeanName" )
 * param $field the field name you want to display errors for (if any)
 *
 *#
#macro( showerror $errors $field )
    #if( $errors )
        #if( $errors.getFieldErrors( $field ))
            #foreach($err in $errors.getFieldErrors( $field ))
                <span class="fieldErrorText">$rc.getMessage($err)</span><br />
            #end
```

```
        #end
    #end
#end
```

12.4.6.3. Associating error messages with the HTML field

Finally, in your html forms, you can use code similar to the following to display bound error messages for each input field.

```
## set the following variable once somewhere at the top of
## the velocity template
#set ($errors=$rc.getErrors("commandBean"))
<html>
...
<form ...>
    <input name="query" value="$!commandBean.query"><br>
    #showerror($errors "query")
</form>
...
</html>
```

12.4.7. 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.jar
          |
          +- velocity
          |   |
          |   +- VM_global_library.vm
          |   +- mainTemplate.vm
          |   +- secondaryTemplate.vm
          |
          +- frontcontroller-servlet.xml
          +- frontcontroller-views.xml
          +- velocity.properties
```

12.5. XSLT Views

XSLT is a transformation language for XML and is popular as a view technology within web applications. XSLT can be a good choice as a view technology if your application naturally deals with XML, or if your model can easily be converted to XML. The following section shows how to produce an XML document as model data and have it transformed with XSLT in a Spring application.

12.5.1. My First Words

This example is a trivial Spring application that creates a list of words in the Controller and adds them to the model map. The map is returned along with the view name of our XSLT view. See Section 11.3, “Controllers”

for details of Spring `Controllers`. The XSLT view will turn the list of words into a simple XML document ready for transformation.

12.5.1.1. Bean definitions

Configuration is standard for a simple Spring application. The dispatcher servlet config file contains a reference to a `ViewResolver`, URL mappings and a single controller bean..

```
<bean id="homeController"class="xslt.HomeController"/>
```

..that implements our word generation 'logic'.

12.5.1.2. Standard MVC controller code

The controller logic is encapsulated in a subclass of AbstractController, with the handler method being defined like so..

```
protected ModelAndView handleRequestInternal(
    HttpServletRequest req,
    HttpServletResponse resp)
    throws Exception {

    Map map = new HashMap();
    List wordList = new ArrayList();

    wordList.add("hello");
    wordList.add("world");

    map.put("wordList", wordList);

    return new ModelAndView("home", map);
}
```

So far we've done nothing that's XSLT specific. The model data has been created in the same way as you would for any other Spring MVC application. Depending on the configuration of the application now, that list of words could be rendered by JSP/JSTL by having them added as request attributes, or they could be handled by Velocity by adding the object to the VelocityContext. In order to have XSLT render them, they of course have to be converted into an XML document somehow. There are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

12.5.1.3. Convert the model data to XML

In order to create a DOM document from our list of words or any other model data, we subclass `org.springframework.web.servlet.view.xslt.AbstractXsltView`. In doing so, we must implement the abstract method `createDomNode()`. The first parameter passed to this method is our model Map. Here's the complete listing of the HomePage class in our trivial word application - it uses JDOM to build the XML document before converting it to the required W3C Node, but this is simply because I find JDOM (and Dom4J) easier API's to handle than the W3C API.

```

package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

    protected Node createDomNode(
        Map model, String rootName, HttpServletRequest req, HttpServletResponse res
    ) throws Exception {

```

```
        org.jdom.Document doc = new org.jdom.Document();
        Element root = new Element(rootName);
        doc.setRootElement(root);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element e = new Element("word");
            e.setText(nextWord);
            root.addContent(e);
        }

        // convert JDOM doc to a W3C Node and return
        return new DOMOutputter().output( doc );
    }

}
```

12.5.1.3.1. Adding stylesheet parameters

A series of parameter name/value pairs can optionally be defined by your subclass which will be added to the transformation object. The parameter names must match those defined in your XSLT template declared with `<xsl:param name="myParam">defaultValue</xsl:param>` To specify the parameters, override the method `getParameters()` from AbstractXsltView and return a `Map` of the name/value pairs.

12.5.1.3.2. Formatting dates and currency

Unlike JSTL and Velocity, XSLT has relatively poor support for locale based currency and date formatting. In recognition of the fact, Spring provides a helper class that you can use from within your `createDomNode()` methods to get such support. See the javadocs for `org.springframework.web.servlet.view.xslt.FormatHelper`

12.5.1.4. Defining the view properties

The views.properties file (or equivalent xml definition if you're using an XML based view resolver as we did in the Velocity examples above) looks like this for the one-view application that is 'My First Words'..

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words
```

Here, you can see how the view is tied in with the HomePage class just written which handles the model domification in the first property '.class'. The stylesheetLocation property obviously points to the XSLT file which will handle the XML transformation into HTML for us and the final property '.root' is the name that will be used as the root of the XML document. This gets passed to the HomePage class above in the second parameter to the `createDomNode` method.

12.5.1.5. Document transformation

Finally, we have the XSLT code used for transforming the above document. As highlighted in the views.properties file, it is called `home.xslt` and it lives in the war file under `WEB-INF/xsl`.

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text/html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>
```

```
                <h1>My First Words</h1>
                <xsl:for-each select="wordList/word">
                    <xsl:value-of select="."/><br />
                </xsl:for-each>

            </body>
        </html>
    </xsl:template>

</xsl:stylesheet>
```

12.5.2. Summary

A summary of the files discussed and their location in the WAR file is shown in the simplified WAR structure below.

```
ProjectRoot
  |
  +- WebContent
      |
      +- WEB-INF
          |
          +- classes
          |    |
          |    +- xslt
          |    |   |
          |    |   +- HomePageController.class
          |    |   +- HomePage.class
          |    |
          |    +- views.properties
          |
          +- lib
          |   |
          |   +- spring.jar
          |
          +- xsl
          |   |
          |   +- home.xslt
          |
          +- frontcontroller-servlet.xml
```

You will also need to ensure that an XML parser and an XSLT engine are available on the classpath. JDK 1.4 provides them by default, and most J2EE containers will also make them available by default, but it's a possible source of errors to be aware of.

12.6. Document views (PDF/Excel)

12.6.1. Introduction

Returning an HTML page isn't always the best way for the user to view the model output, and Spring makes it simple to generate a PDF document or an Excel spreadsheet dynamically from the model data. The document is the view and will be streamed from the server with the corrdct content type to (hopefully) enable the client PC to run the spreadsheet or PDF viewer application in response.

In order to use Excel views, you need to add the 'poi' library to your classpath, and for PDF generation, the iText.jar. Both are included in the main Spring distribution.

12.6.2. Configuration and setup

Document based views are handled in an almost identical fashion to XSLT views, and the following sections build upon the previous one by demonstrating how the same controller used in the XSLT example is invoked to render the same model as both a PDF document and an Excel spreadsheet (which can also be viewed or manipulated in Open Office).

12.6.2.1. Document view definitions

Firstly, let's amend the views.properties file (or xml equivalent) and add a simple view definition for both document types. The entire file now looks like this with the XSLT view shown from earlier..

```
home.class=xslt.HomePage
home.stylesheetLocation=/WEB-INF/xsl/home.xslt
home.root=words

xl.class=excel.HomePage

pdf.class=pdf.HomePage
```

If you want to start with a template spreadsheet to add your model data to, specify the location as the 'url' property in the view definition

12.6.2.2. Controller code

The controller code we'll use remains exactly the same from the XSLT example earlier other than to change the name of the view to use. Of course, you could be clever and have this selected based on a URL parameter or some other logic - proof that Spring really is very good at decoupling the views from the controllers!

12.6.2.3. Subclassing for Excel views

Exactly as we did for the XSLT example, we'll subclass suitable abstract classes in order to implement custom behaviour in generating our output documents. For Excel, this involves writing a subclass of `org.springframework.web.servlet.view.document.AbstractExcelView` and implementing the `buildExcelDocument`

Here's the complete listing for our Excel view which displays the word list from the model map in consecutive rows of the first column of a new spreadsheet..

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(
        Map model,
        HSSFWorkbook wb,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        HSSFSheet sheet;
        HSSFRow sheetRow;
        HSSFCell cell;

        // Go to the first sheet
        // getSheetAt: only if wb is created from an existing document
        //sheet = wb.getSheetAt( 0 );
        sheet = wb.createSheet("Spring");
        sheet.setDefaultColumnWidth((short)12);

        // write a text at A1
        cell = getCell( sheet, 0, 0 );
        setText(cell,"Spring-Excel test");
```

```
        List words = (List ) model.get("wordList");
        for (int i=0; i < words.size(); i++) {
            cell = getCell( sheet, 2+i, 0 );
            setText(cell, (String) words.get(i));

        }
    }
}
```

If you now amend the controller such that it returns `xl` as the name of the view (`return new ModelAndView("xl", map);`) and run your application again, you should find that the Excel spreadsheet is created and downloaded automagically when you request the same page as before.

12.6.2.4. Subclassing for PDF views

The PDF version of the word list is even simpler. This time, the class extends `org.springframework.web.servlet.view.document.AbstractPdfView` and implements the `buildPdfDocument()` method as follows..

```
package pdf;

// imports omitted for brevity

public class PDFPage extends AbstractPdfView {

    protected void buildPdfDocument(
        Map model,
        Document doc,
        PdfWriter writer,
        HttpServletRequest req,
        HttpServletResponse resp)
        throws Exception {

        List words = (List) model.get("wordList");

        for (int i=0; i<words.size(); i++)
            doc.add( new Paragraph((String) words.get(i)));

    }
}
```

Once again, amend the controller to return the `pdf` view with a `return new ModelAndView("pdf", map);` and reload the URL in your application. This time a PDF document should appear listing each of the words in the model map.

12.7. Tapestry

Tapestry is a powerful, component-oriented web application framework from Apache's Jakarta project (http://jakarta.apache.org/tapestry). Spring Framework is a powerful J2EE application framework which is built around the concept of a lightweight container. While Spring has its own powerful web ui layer, there are a number of unique advantages to building a J2EE application using a combination of Tapestry for the web ui, and the Spring container for the lower layers. This document attempts to detail a few best practices for combining these two frameworks. It is expected that you are relatively familiar with both Tapestry and Spring Framework basics, so they will not be explained here. General introductory documentation for both Tapestry and Spring Framework are available on their respective web sites.

12.7.1. Architecture

A typical layered J2EE application built with Tapestry and Spring will consist of a top UI layer built with

Tapestry, and a number of lower layers, hosted out of one or more Spring Application Contexts.

- *User Interface Layer:*

  - concerned with the user interface

  - contains some application logic

  - provided by Tapestry

  - aside from providing UI via Tapestry, code in this layer does its work via objects which implement interfaces from the Service Layer. The actual objects which implement these service layer interfaces are obtained from a Spring Application Context.

- *Service Layer:*

  - application specific 'service' code

  - works with domain objects, and uses the Mapper API to get those domain objects into and out of some sort of repository (database)

  - hosted in one or more Spring contexts

  - code in this layer manipulates objects in the domain model, in an application specific fashion. It does its work via other code in this layer, and via the Mapper API. An object in this layer is given the specific mapper implementations it needs to work with, via the Spring context.

  - since code in this layer is hosted in the Spring context, it may be transactionally wrapped by the Spring context, as opposed to managing its own transactions

- *Domain Model:*

  - domain specific object hierarchy, which deals with data and logic specific to this domain

  - although the domain object hierarchy is built with the idea that it is persisted somehow and makes some general concessions to this (for example, bidirectional relationships), it generally has no knowledge of other layers. As such, it may be tested in isolation, and used with different mapping implementations for production vs. testing.

  - these objects may be standalone, or used in conjunction with a Spring application context to take advantage of some of the benefits of the context, e.g., isolation, inversion of control, different strategy implementations, etc.

- *Data Source Layer:*

  - Mapper API (also called Data Access Objects): an API used to persist the domain model to a repository of some sort (generally a DB, but could be the filesystem, memeory, etc.)

  - Mapper API implementations: one or more specific implementations of the Mapper API, for example, a Hibernate-specific mapper, a JDO-specific mapper, JDBC-specific mapper, or a memory mapper.

  - mapper implementations live in one or more Spring Application Contexts. A service layer object is given the mapper objects it needs to work with via the context.

- *Database, filesystem, or other repositories:*

- objects in the domain model are stored into one or more repositories via one or more mapper implementations

- a repository may be very simpe (e.g. filesystem), or may have its own representation of the data from the domain model (i.e. a schema in a db). It does not know about other layers howerver.

12.7.2. Implementation

The only real question (which needs to be addressed by this document), is how Tapestry pages get access to service implementations, which are simply beans defined in an instance of the Spring Application Context.

12.7.2.1. Sample application context

Assume we have the following simple Application Context definition, in xml form:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <!-- ========================= GENERAL DEFINITIONS ========================= -->

    <!-- ========================= PERSISTENCE DEFINITIONS ========================= -->

    <!-- the DataSource -->
    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName"><value>java:DefaultDS</value></property>
        <property name="resourceRef"><value>false</value></property>
    </bean>

    <!-- define a Hibernate Session factory via a Spring LocalSessionFactoryBean -->
    <bean id="hibSessionFactory"
        class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource"><ref bean="dataSource"/></property>
    </bean>

    <!--
     - Defines a transaction manager for usage in business or data access objects.
     - No special treatment by the context, just a bean instance available as reference
     - for business objects that want to handle transactions, e.g. via TransactionTemplate.
     -->
    <bean id="transactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager">
    </bean>

    <bean id="mapper"
        class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
        <property name="sessionFactory"><ref bean="hibSessionFactory"/></property>
    </bean>

    <!-- ========================= BUSINESS DEFINITIONS ========================= -->

    <!-- AuthenticationService, including tx interceptor -->
    <bean id="authenticationServiceTarget"
        class="com.whatever.services.service.user.AuthenticationServiceImpl">
        <property name="mapper"><ref bean="mapper"/></property>
    </bean>
    <bean id="authenticationService"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager"><ref bean="transactionManager"/></property>
        <property name="target"><ref bean="authenticationServiceTarget"/></property>
        <property name="proxyInterfacesOnly"><value>true</value></property>
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>
```

```
    <!-- UserService, including tx interceptor -->
    <bean id="userServiceTarget"
        class="com.whatever.services.service.user.UserServiceImpl">
        <property name="mapper"><ref bean="mapper"/></property>
    </bean>
    <bean id="userService"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager"><ref bean="transactionManager"/></property>
        <property name="target"><ref bean="userServiceTarget"/></property>
        <property name="proxyInterfacesOnly"><value>true</value></property>
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

</beans>
```

Inside the Tapestry application, we need to load this application context, and allow Tapestry pages to get the authenticationService and userService beans, which implement the AuthenticationService and UserService interfaces, respectively.

12.7.2.2. Obtaining beans in Tapestry pages

At this point, the application context is available to a web application by calling Spring's static utility function `WebApplicationContextUtils.getApplicationContext(servletContext)`, where servletContext is the standard `ServletContext` from the J2EE Servlet specification. As such, one simple mechanism for a page to get an instance of the UserService, for example, would be with code such as:

```
    WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
        getRequestCycle().getRequestContext().getServlet().getServletContext());
    UserService userService = appContext.getBean("userService");
    ... some code which uses UserService
```

This mechanism does work. It can be made a lot less verbose by encapsulating most of the functionality in a method in the base class for the page or component. However, in some respects it goes against the Inversion of Control approach which Spring encourages, which is being used in other layers of this app, in that ideally you would like the page to not have to ask the context for a specific bean by name, and in fact, the page would ideally not know about the context at all.

Luckily, there is a mechanism to allow this. We rely upon the fact that Tapestry already has a mechanism to declaratively add properties to a page, and it is in fact the preferred approach to manage all properties on a page in this declarative fashion, so that Tapestry can properly manage their lifecycle as part of the page and component lifecycle.

12.7.2.3. Exposing the application context to Tapestry

First we need to make the `ApplicationContext` available to the Tapestry page or Component without having to have the `ServletContext`; this is because at the stage in the page's/component's lifecycle when we need to access the `ApplicationContext`, the `ServletContext` won't be easily available to the page, so we can't use `WebApplicationContextUtils.getApplicationContext(servletContext)` directly. One way is by defining a custom version of the Tapestry IEngine which exposes this for us:

```
package com.whatever.web.xportal;
...
import ...
...
public class MyEngine extends org.apache.tapestry.engine.BaseEngine {

    public static final String APPLICATION_CONTEXT_KEY = "appContext";
```

```
    /**
     * @see org.apache.tapestry.engine.AbstractEngine#setupForRequest(org.apache.tapestry.request.RequestContext
     */
    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);

        // insert ApplicationContext in global, if not there
        Map global = (Map) getGlobal();
        ApplicationContext ac = (ApplicationContext) global.get(APPLICATION_CONTEXT_KEY);
        if (ac == null) {
            ac = WebApplicationContextUtils.getWebApplicationContext(
                context.getServlet().getServletContext()
            );
            global.put(APPLICATION_CONTEXT_KEY, ac);
        }
    }
}
```

This engine class places the Spring Application Context as an attribute called "appContext" in this Tapestry app's 'Global' object. Make sure to register the fact that this special IEngine instance should be used for this Tapestry application, with an entry in the Tapestry application definition file. For example:

```
file: xportal.application:

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">
<application
    name="Whatever xPortal"
    engine-class="com.whatever.web.xportal.MyEngine">
</application>
```

12.7.2.4. Component definition files

Now in our page or component definition file (*.page or *.jwc), we simply add property-specification elements to grab the beans we need out of the ApplicationContext, and create page or component properties for them. For example:

```
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>
```

The OGNL expression inside the property-specification specifies the initial value for the property, as a bean obtained from the context. The entire page definition might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="com.whatever.web.xportal.pages.Login">

    <property-specification name="username" type="java.lang.String"/>
    <property-specification name="password" type="java.lang.String"/>
    <property-specification name="error" type="java.lang.String"/>
    <property-specification name="callback" type="org.apache.tapestry.callback.ICallback" persistent="yes"/>
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>
```

```
    <bean name="delegate" class="com.whatever.web.xportal.PortalValidationDelegate"/>

    <bean name="validator" class="org.apache.tapestry.valid.StringValidator" lifecycle="page">
        <set-property name="required" expression="true"/>
        <set-property name="clientScriptingEnabled" expression="true"/>
    </bean>

    <component id="inputUsername" type="ValidField">
        <static-binding name="displayName" value="Username"/>
        <binding name="value" expression="username"/>
        <binding name="validator" expression="beans.validator"/>
    </component>

    <component id="inputPassword" type="ValidField">
        <binding name="value" expression="password"/>
       <binding name="validator" expression="beans.validator"/>
       <static-binding name="displayName" value="Password"/>
       <binding name="hidden" expression="true"/>
    </component>

</page-specification>
```

12.7.2.5. Adding abstract accessors

Now in the Java class definition for the page or component itself, all we need to do is add an abstract getter method for the properties we have defined, to access them. When the page or component is actually loaded by Tapestry, it performs runtime code instrumentation on the classfile to add the properties which have been defined, and hook up the abstract getter methods to the newly created fields. For example:

```
    // our UserService implementation; will come from page definition
    public abstract UserService getUserService();
    // our AuthenticationService implementation; will come from page definition
    public abstract AuthenticationService getAuthenticationService();
```

For completeness, the entire Java class, for a login page in this example, might look like this:

```
package com.whatever.web.xportal.pages;

/**
 *  Allows the user to login, by providing username and password.
 *  After succesfully logging in, a cookie is placed on the client browser
 *  that provides the default username for future logins (the cookie
 *  persists for a week).
 */
public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    /** the key under which the authenticated user object is stored in the visit as */
    public static final String USER_KEY = "user";

    /**
     * The name of a cookie to store on the user's machine that will identify
     * them next time they log in.
     **/
    private static final String COOKIE_NAME = Login.class.getName() + ".username";
    private final static int ONE_WEEK = 7 * 24 * 60 * 60;

    // --- attributes

    public abstract String getUsername();
    public abstract void setUsername(String username);

    public abstract String getPassword();
    public abstract void setPassword(String password);

    public abstract ICallback getCallback();
    public abstract void setCallback(ICallback value);

    public abstract UserService getUserService();

    public abstract AuthenticationService getAuthenticationService();
```

```
    // --- methods

    protected IValidationDelegate getValidationDelegate() {
        return (IValidationDelegate) getBeans().getBean("delegate");
    }

    protected void setErrorField(String componentId, String message) {
        IFormComponent field = (IFormComponent) getComponent(componentId);
        IValidationDelegate delegate = getValidationDelegate();
        delegate.setFormComponent(field);
        delegate.record(new ValidatorException(message));
    }

    /**
     *  Attempts to login.
     *
     *  <p>If the user name is not known, or the password is invalid, then an error
     *  message is displayed.
     *
     **/
    public void attemptLogin(IRequestCycle cycle) {

        String password = getPassword();

        // Do a little extra work to clear out the password.

        setPassword(null);
        IValidationDelegate delegate = getValidationDelegate();

        delegate.setFormComponent((IFormComponent) getComponent("inputPassword"));
        delegate.recordFieldInputValue(null);

        // An error, from a validation field, may already have occured.

        if (delegate.getHasErrors())
            return;

        try {
            User user = getAuthenticationService().login(getUsername(), getPassword());
           loginUser(user, cycle);
        }
        catch (FailedLoginException ex) {
            this.setError("Login failed: " + ex.getMessage());
            return;
        }
    }

    /**
     *  Sets up the {@link User} as the logged in user, creates
     *  a cookie for their username (for subsequent logins),
     *  and redirects to the appropriate page, or
     *  a specified page).
     *
     **/
    public void loginUser(User user, IRequestCycle cycle) {

        String username = user.getUsername();

        // Get the visit object; this will likely force the
        // creation of the visit object and an HttpSession.

        Map visit = (Map) getVisit();
        visit.put(USER_KEY, user);

        // After logging in, go to the MyLibrary page, unless otherwise
        // specified.

        ICallback callback = getCallback();

        if (callback == null)
            cycle.activate("Home");
        else
            callback.performCallback(cycle);

        // I've found that failing to set a maximum age and a path means that
        // the browser (IE 5.0 anyway) quietly drops the cookie.

        IEngine engine = getEngine();
```

```
        Cookie cookie = new Cookie(COOKIE_NAME, username);
        cookie.setPath(engine.getServletPath());
        cookie.setMaxAge(ONE_WEEK);

        // Record the user's username in a cookie

        cycle.getRequestContext().addCookie(cookie);

        engine.forgetPage(getPageName());
    }

    public void pageBeginRender(PageEvent event) {
        if (getUsername() == null)
            setUsername(getRequestCycle().getRequestContext().getCookieValue(COOKIE_NAME));
    }
}
```

12.7.3. Summary

In this example, we've managed to allow service beans defined in the Spring `ApplicationContext` to be provided to the page in a declarative fashion. The page class does not know where the service implementations are coming from, and in fact it is easy to slip in another implementation, for example, during testing. This inversion of control is one of the prime goals and benefits of the Spring Framework, and we have managed to extend it all the way up the J2EE stack in this Tapestry application.

Chapter 13. Accessing and implementing EJBs

As a lightweight container, Spring is often considered an EJB replacement. We do believe that for many if not most applications and use cases, Spring as a container, combined with its rich supporting functionality in the area of transactions, ORM and JDBC access, is a better choice than implementing equivalent functionality via an EJB container and EJBs.

However, it is important to note that using Spring does not prevent you from using EJBs. In fact, Spring makes it much easier to access EJBs and implement EJBs and functionality within them. Additionally, using Spring to access services provided by EJBs allows the implementation of those services to later transparently be switched between local EJB, remote EJB, or POJO (plain java object) variants, without the client code client code having to be changed.

In this chapter, we look at how Spring can help you access and implement EJBs. Spring provides particular value when accessing stateless session beans (SLSBs), so we'll begin by discussing this.

13.1. Accessing EJBs

13.1.1. Concepts

To invoke a method on a local or remote stateless session bean, client code must normally perform a JNDI lookup to obtain the (local or remote) EJB Home object, then use a 'create' method call on that object to obtain the actual (local or remote) EJB object. One or more methods are then invoked on the EJB.

To avoid repeated low-level code, many EJB applications use the Service Locator and Business Delegate patterns. These are better than spraying JNDI lookups throughout client code, but their usual implementations have significant disadvantages. For example:

- Typically code using EJBs depends on Service Locator or Business Delegate singletons, making it hard to test
- In the case of the Service Locator pattern used without a Business Delegate, application code still ends up having to invoke the create() method on an EJB home, and deal with the resulting exceptions. Thus it remains tied to the EJB API and the complexity of the EJB programming model.
- Implementing the Business Delegate pattern typically results in significant code duplication, where we have to write numerous methods that simply call the same method on the EJB.

The Spring approach is to allow the creation and use of proxy objects, normally configured inside a Spring ApplicationContext or BeanFactory, which act as codeless business delegates. You do not need to write another Service Locator, another JNDI lookup, or duplicate methods in a hand-coded Business Delegate unless you’re adding real value.

13.1.2. Accessing local SLSBs

Assume that we have a web controller that needs to use a local EJB. We’ll follow best practice and use the EJB Business Methods Interface pattern, so that the EJB’s local interface extends a non EJB-specific business methods interface. Let’s call this business methods interface MyComponent.

```
public interface MyComponent {
    ...
```

```
}
```

(One of the main reasons to the Business Methods Interface pattern is to ensure that synchronization between method signatures in local interface and bean implementation class is automatic. Another reason is that it later makes it much easier for us to switch to a POJO (plain java object) implementation of the service if it makes sense to do so) Of course we'll also need to implement the local home interface and provide a bean implementation class that implements SessionBean and the MyComponent business methods interface. Now the only Java coding we'll need to do to hook up our web tier controller to the EJB implementation is to expose a setter method of type MyComponent on the controller. This will save the reference as an instance variable in the controller:

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

We can subsequently use this instance variable in any business method in the controller. Now assuming we are obtaining our controller object out of a Spring ApplicationContext or BeanFactory, we can in the same context configure a `LocalStatelessSessionProxyFactoryBean` instance, which will be EJB proxy object. The configuration of the proxy, and setting of the `myComponent` property of the controller is done with a configuration entry such as:

```
<bean id="myComponent"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
    <property name="jndiName"><value>myComponent</value></property>
    <property name="businessInterface"><value>com.mycom.MyComponent</value></property>
</bean>

<bean id="myController" class = "com.mycom.myController">
    <property name="myComponent"><ref bean="myComponent"/></property>
</bean>
```

There's a lot of magic happening behind the scenes, courtesy of the Spring AOP framework, although you aren't forced to work with AOP concepts to enjoy the results. The `myComponent` bean definition creates a proxy for the EJB, which implements the business method interface. The EJB local home is cached on startup, so there's only a single JNDI lookup. Each time the EJB is invoked, the proxy invokes the create() method on the local EJB and invokes the corresponding business method on the EJB.

The `myController` bean definition sets the `myController` property of the controller class to this proxy.

This EJB access mechanism delivers huge simplification of application code: The web tier code (or other EJB client code) has no dependence on the use of EJB. If we want to replace this EJB reference with a POJO or a mock object or other test stub, we could simply change the myComponent bean definition without changing a line of Java code Additonally, we haven't had to write a single line of JNDI lookup or other EJB plumbing code as part of our application.

Benchmarks and experience in real applications indicate that the performance overhead of this approach (which involves reflective invocation of the target EJB) is minimal, and undetectable in typical use. Remember that we don't want to make fine-grained calls to EJBs anyway, as there's a cost associated with the EJB infrastructure in the application server.

There is one caveat wIth regards the JNDI lookup. In a bean container, this class is normally best used as a singleton (there simply is no reason to make it a prototype). However, if that bean container pre-instantiates singletons (as do the XML ApplicationContext variants) you may have a problem if the bean container is loaded before the EJB container loads the target EJB. That is because the JNDI lookup will be performed in the

init method of this class and cached, but the EJB will not have been bound at the target location yet. The solution is to not pre-instantiate this factory object, but allow it to be created on first use. In the XML containers, this is controlled via the `lazy-init` attribute.

Although this will not be of interest to the majority of Spring users, those doing programmatic AOP work with EJBs may want to look at `LocalSlsbInvokerInterceptor`.

13.1.3. Accessing remote SLSBs

Accessing remote EJBs is essentially identical to accessing local EJBs, except that the `SimpleRemoteStatelessSessionProxyFactoryBean` is used. Of course, with or without Spring, remote invocation semantics apply; a call to a method on an object in another VM in another computer does sometimes have to be treated differently in terms of usage scenarios and failure handling.

Spring's EJB client support adds one more advantage over the non-Spring approach. Normally it is problematic for EJB client code to be easily switched back and forth between calling EJBs locally or remotely. This is because the remote interface methods must declare that they throw `RemoteException`, and client code must deal with this, while the local interface methods don't. Client code wrCCitten for local EJBs which needs to be moved to remote EJBs typically has to be modified to add handling for the remote exceptions, and client code written for remote EJBs which needs to be moved to local EJBs, can either stay the same but do a lot of unnecessary handling of remote excep

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

tions, or needs to be modified to remove that code. With the Spring remote EJB proxy, you can instead not declare any thrown `RemoteException` in your Business Method Interface and implementing EJB code, have a remote interface which is identical except that it does throw `RemoteException`, and rely on the proxy to dynamically treat the two interfaces as if they were the same. That is, client code does not have to deal with the checked `RemoteException` class. Any actual `RemoteException` that is thrown during the EJB invocation will be rethrown as the non-checked `RemoteAccessException` class, which is a subclass of `RuntimeException`. The target service can then be switched at will between a local EJB or remote EJB (or even plain Java object) implementation, without the client code knowing or caring. Of course, this is optional; there is nothing stopping you from declaring `RemoteExceptions` in your business interface.

13.2. Using Spring convenience EJB implementation classes

Spring also provides convenience classes to help you implement EJBs. These are designed to encourage the good practTo ice of putting business logic behind EJBs in POJOs, leaving EJBs responsible for transaction demarcation and (optionally) remoting.

To implement a Stateless or Stateful session bean, or Message Driven bean, you derive your implementation class from `AbstractStatelessSessionBean`, `AbstractStatefulSessionBean`, and `AbstractMessageDrivenBean`/`AbstractJmsMessageDrivenBean`, respectively.

Consider an example Stateless Session bean which actually delgates the implementation to a plain java service object. We have the business interface:

```
public interface MyComponent {
    public void myMethod(...);
    ...
}
```

We have the plain java implementation object:

```
public class MyComponentImpl implements MyComponent {
    public String myMethod(...) {
        ...
    }
    ...
}
```

And finally the Stateless Session Bean itself:

```
public class MyComponentEJB implements extends AbstractStatelessSessionBean
        implements MyComponent {

    MyComponent _myComp;

    /**
     * Obtain our POJO service object from the BeanFactory/ApplicationContext
     * @see org.springframework.ejb.support.AbstractStatelessSessionBean#onEjbCreate()
     */
    protected void onEjbCreate() throws CreateException {
        _myComp = (MyComponent) getBeanFactory().getBean(
            ServicesConstants.CONTEXT_MYCOMP_ID);
    }

    // for business method, delegate to POJO service impl.
    public String myMethod(...) {
        return _myComp.myMethod(...);
    }
    ...
}
```

The Spring EJB support base classes will by default create and load a BeanFactory (or in this case, its ApplicationContext subclass) as part of their lifecycle, which is then available to the EJB (for example, as used in the code above to obtain the POJO service object). The loading is done via a strategy object which is a subclass of `BeanFactoryLocator`. The actual implementation of `BeanFactoryLocator` used by default is `ContextJndiBeanFactoryLocator`, which creates the ApplicationContext from a resource locations specified as a JNDI environment variable (in the case of the EJB classes, at `java:comp/env/ejb/BeanFactoryPath`). If there is a need to change the BeanFactory/ApplicationContext loading strategy, the default `BeanFactoryLocator` implementation used may be overriden by calling the `setBeanFactoryLocator()` method, either in `setSessionContext()`, or in the actual constructor of the EJB. Please see the JavaDocs for more details.

As described in the JavaDocs, Stateful Session beans expecting to be passivated and reactivated as part of their lifecycle, and which use a non-serializable BeanFactory/ApplicationContext instance (which is the normal case) will have to manually call `unloadBeanFactory()` and `loadBeanFactory` from `ejbPassivate` and `ejbActivate`, respectively, to unload and reload the BeanFactory on passivation and activation, since it can not be saved by the EJB container.

The default usage of `ContextJndiBeanFactoryLocator` to load an ApplicationContext for the use of the EJB is adequate for some situations. However, it is problematic when the ApplicationContext is loading a number of beans, or the initializaiton of those beans is time consuming or memory intensive (such as a Hibernate SessionFactory initialization, for example), since every EJB will have their own copy. In this case, the use may want to override the defualt `ContextJndiBeanFactoryLocator` usage and use another `BeanFactoryLocator` variant, such as `ContextSingletonBeanFactoryLocator`, which can load and use a shared BeanFactory or ApplicationContext to be used by multiple EJBs or other clients. Doing this is relatively simple, by adding code similar to this to the EJB:

```
   /**
    * Override default BeanFactoryLocator implementation
    *
    * @see javax.ejb.SessionBean#setSessionContext(javax.ejb.SessionContext)
```

```
     */
    public void setSessionContext(SessionContext sessionContext) {
        super.setSessionContext(sessionContext);
        setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());
        setBeanFactoryLocatorKey(ServicesConstants.PRIMARY_CONTEXT_ID);
    }
```

Please see the respective JavaDocs for `BeanFactoryLocator` and `ContextSingletonBeanFactoryLocator` for more information on their usage.

Chapter 14. Sending Email with Spring mail abstraction layer

14.1. Introduction

Spring provides a higher level of abstraction for sending electronic mail which shields the user from the specifics of underlying mailing system and is responsible for a low level resource handling on behalf of the client.

14.2. Spring mail abstraction structure

The main package of Spring mail abstraction layer is `org.springframework.mail` package. It contains central interface for sending emails called `MailSender` and the *value object* which encapsulates properties of a simple mail such as *from*, *to*, *cc*, *subject*, *text* called `SimpleMailMessage`. This package also conatins a hierarchy of checked exceptions which provide a higher level of abstraction over the lower level mail system expetions with the root exception being `MailException`. Please refer to JavaDocs for more information on mail exception hierarchy.

Spring also provides a subinterface of `MailSender` for specialized *JavaMail* features such as MIME messages, namely `org.springframework.mail.javamail.JavaMailSender` It also provides a callback interface for praparation of JavaMail MIME messages, namely `org.springframework.mail.javamail.MimeMessagePreparator`

MailSender:

```
public interface MailSender {

    /**
     * Send the given simple mail message.
     * @param simpleMessage message to send
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(SimpleMailMessage simpleMessage) throws MailException;

    /**
     * Send the given array of simple mail messages in batch.
     * @param simpleMessages messages to send
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(SimpleMailMessage[] simpleMessages) throws MailException;

}
```

JavaMailSender:

```
public interface JavaMailSender extends MailSender {

    /**
     * Create a new JavaMail MimeMessage for the underlying JavaMail Session
     * of this sender. Needs to be called to create MimeMessage instances
     * that can be prepared by the client and passed to send(MimeMessage).
     * @return the new MimeMessage instance
     * @see #send(MimeMessage)
     * @see #send(MimeMessage[])
     */
    public MimeMessage createMimeMessage();

    /**
     * Send the given JavaMail MIME message.
     * The message needs to have been created with createMimeMessage.
```

```
     * @param mimeMessage message to send
     * @throws MailException in case of message, authentication, or send errors
     * @see #createMimeMessage
     */
    public void send(MimeMessage mimeMessage) throws MailException;

    /**
     * Send the given array of JavaMail MIME messages in batch.
     * The messages need to have been created with createMimeMessage.
     * @param mimeMessages messages to send
     * @throws MailException in case of message, authentication, or send errors
     * @see #createMimeMessage
     */
    public void send(MimeMessage[] mimeMessages) throws MailException;

    /**
     * Send the JavaMail MIME message prepared by the given MimeMessagePreparator.
     * Alternative way to prepare MimeMessage instances, instead of createMimeMessage
     * and send(MimeMessage) calls. Takes care of proper exception conversion.
     * @param mimeMessagePreparator the preparator to use
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(MimeMessagePreparator mimeMessagePreparator) throws MailException;

    /**
     * Send the JavaMail MIME messages prepared by the given MimeMessagePreparators.
     * Alternative way to prepare MimeMessage instances, instead of createMimeMessage
     * and send(MimeMessage[]) calls. Takes care of proper exception conversion.
     * @param mimeMessagePreparators the preparator to use
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(MimeMessagePreparator[] mimeMessagePreparators) throws MailException;

}
```

MimeMessagePreparator:

```
public interface MimeMessagePreparator {

    /**
     * Prepare the given new MimeMessage instance.
     * @param mimeMessage the message to prepare
     * @throws MessagingException passing any exceptions thrown by MimeMessage
     * methods through for automatic conversion to the MailException hierarchy
     */
    void prepare(MimeMessage mimeMessage) throws MessagingException;

}
```

14.3. Using Spring mail abstraction

Let's assume there is a businness interface called `OrderManager`

```
public interface OrderManager {

    void placeOrder(Order order);
}
```

and there is a use case that says that an email message with order number would need to be generated and sent to a customer placing that order. So for this purpose we want to use `MailSender` and `SimpleMailMessage`

Please note that as usual, we work with interfaces in the business code and let Spring IoC container take care of wiring of all the collaborators for us.

Here is the implementation of `OrderManager`

```
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;
```

```
public class OrderManagerImpl implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage message;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setMessage(SimpleMailMessage message) {
        this.message = message;
    }

    public void placeOrder(Order order) {

        //... * Do the businness calculations....
        //... * Call the collaborators to persist the order

        //Create a threadsafe "sandbox" of the message
        SimpleMailMessage msg = new SimpleMailMessage(this.message);
        msg.setTo(order.getCustomer().getEmailAddress());
        msg.setText(
            "Dear "
                + order.getCustomer().getFirstName()
                + order.getCustomer().getLastName()
                + ", thank you for placing order. Your order number is "
                + order.getOrderNumber());
        try{
            mailSender.send(msg);
        }
        catch(MailException ex) {
            //log it and go on
            System.err.println(ex.getMessage());
        }
    }
}
```

Here is what the bean definitions for the code above would look like:

```
<bean id="mailSender"
      class="org.springframework.mail.javamail.JavaMailSenderImpl">
    <property name="host"><value>mail.mycompany.com</value></property>
</bean>

<bean id="mailMessage"
      class="org.springframework.mail.SimpleMailMessage">
    <property name="from"><value>customerservice@mycompany.com</value></property>
    <property name="subject"><value>Your order</value></property>
</bean>

<bean id="orderManager"
      class="com.mycompany.businessapp.support.OrderManagerImpl">
    <property name="mailSender"><ref bean="mailSender"/></property>
    <property name="message"><ref bean="mailMessage"/></property>
</bean>
```

Here is the implementation of `OrderManager` using `MimeMessagePreparator` callback interface. Please note that the mailSender property is of type `JavaMailSender` in this case in order to be able to use JavaMail MimeMessage:

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class OrderManagerImpl implements OrderManager {
    private JavaMailSender mailSender;
```

```
    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {

        //... * Do the businness calculations....
        //... * Call the collaborators to persist the order


        MimeMessagePreparator preparator = new MimeMessagePreparator() {
            public void prepare(MimeMessage mimeMessage) throws MessagingException {
                mimeMessage.setRecipient(Message.RecipientType.TO,
                        new InternetAddress(order.getCustomer().getEmailAddress()));
                mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));
                mimeMessage.setText(
                    "Dear "
                        + order.getCustomer().getFirstName()
                        + order.getCustomer().getLastName()
                        + ", thank you for placing order. Your order number is "
                        + order.getOrderNumber());
            }
        };
        try{
            mailSender.send(preparator);
        }
        catch(MailException ex) {
            //log it and go on
            System.err.println(ex.getMessage());
        }
    }
}
```

If you want to use JavaMail MimeMessage to the full power, the `MimeMessagePreparator` is available at your fingertips.

Please note that the mail code is a crosscutting concern and is a perfect candidate for refactoring into a custom SpringAOP advice, which then could easily be applied to OrderManager target. Please see the AOP chapter.

14.3.1. Pluggable MailSender implementations

Spring comes with two MailSender implementations out of the box - the JavaMail implementation and the implementation on top of Jason Hunter's *MailMessage* class that's included in http://servlets.com/cos (com.oreilly.servlet). Please refer to JavaDocs for more information.

Appendix A. Spring's `beans.dtd`

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
    Spring XML BeanFactory DTD.
    Authors: Rod Johnson, Juergen Hoeller

    This defines a simple and consistent way of creating a namespace
    of JavaBeans configured by a Spring XmlBeanFactory.

    This document type is used by most Spring functionality, including
    web application contexts, which are based on bean factories.

    Each <bean> element in this document defines a JavaBean.
    Typically the bean class is specified, along with JavaBean properties.

    Bean instances can be "singletons" (shared instances) or "prototypes"
    (independent instances).

    References among beans are supported, i.e. setting a JavaBean property
    to refer to another bean in the same factory or an ancestor factory.

    As alternative to bean references, "inner bean definitions" can be used.
    Singleton flags and names of such "inner beans" are always ignored:
    Inner beans are anonymous prototypes.

    There is also support for lists, maps, and java.util.Properties types.

    As the format is simple, a DTD is sufficient, and there's no need
    for a schema at this point.

    XML documents that conform to this DTD should declare the following doctype:
    <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

    $Id: dtd.xml,v 1.4 2004/03/22 01:44:23 trisberg Exp $
-->

<!--
    Element containing informative text describing the purpose of the enclosing
    element. Always optional.
    Used primarily for user documentation of XML bean definition documents.
-->
<!ELEMENT description (#PCDATA)>

<!--
    The document root.
    At least one bean definition is required.
-->
<!ELEMENT beans (
        description?,
        bean+
)>

<!--
    Default values for all bean definitions. Can be overridden at
    the "bean" level. See those attribute definitions for details.
-->
<!ATTLIST beans default-lazy-init (true | false) "false">
<!ATTLIST beans default-dependency-check (none | objects | simple | all) "none">
<!ATTLIST beans default-autowire (no | byName | byType | constructor | autodetect) "no">

<!--
    Defines a single named bean.
-->
<!ELEMENT bean (
        description?,
        (constructor-arg | property)*
)>

<!--
    Beans can be identified by an id, to enable reference checking.
    There are constraints on a valid XML id: if you want to reference your bean
```

```
    in Java code using a name that's illegal as an XML id, use the optional
    "name" attribute. If neither given, the bean class name is used as id.
-->
<!ATTLIST bean id ID #IMPLIED>

<!--
    Optional. Can be used to create one or more aliases illegal in an id.
    Multiple aliases can be separated by any number of spaces or commas.
-->
<!ATTLIST bean name CDATA #IMPLIED>

<!--
    Each bean definition must specify the FQN of the class,
    or the name of the parent bean from which the class can be worked out.

    Note that a child bean definition that references a parent will just
    add respectively override property values and be able to change the
    singleton status. It will inherit all of the parent's other parameters
    like lazy initialization or autowire settings.
-->
<!ATTLIST bean class CDATA #IMPLIED>
<!ATTLIST bean parent CDATA #IMPLIED>

<!--
    Is this bean a "singleton" (one shared instance, which will
    be returned by all calls to getBean() with the id),
    or a "prototype" (independent instance resulting from each call to
    getBean(). Default is singleton.

    Singletons are most commonly used, and are ideal for multi-threaded
    service objects.
-->
<!ATTLIST bean singleton (true | false) "true">

<!--
    If this bean should be lazily initialized.
    If false, it will get instantiated on startup by bean factories
    that perform eager initialization of singletons.
-->
<!ATTLIST bean lazy-init (true | false | default) "default">

<!--
    Optional attribute controlling whether to "autowire" bean properties.
    This is an automagical process in which bean references don't need to be coded
    explicitly in the XML bean definition file, but Spring works out dependencies.

    There are 5 modes:

    1. "no"
    The traditional Spring default. No automagical wiring. Bean references
    must be defined in the XML file via the <ref> element. We recommend this
    in most cases as it makes documentation more explicit.

    2. "byName"
    Autowiring by property name. If a bean of class Cat exposes a dog property,
    Spring will try to set this to the value of the bean "dog" in the current factory.

    3. "byType"
    Autowiring if there is exactly one bean of the property type in the bean factory.
    If there is more than one, a fatal error is raised, and you can't use byType
    autowiring for that bean. If there is none, nothing special happens - use
    dependency-check="objects" to raise an error in that case.

    4. "constructor"
    Analogous to "byType" for constructor arguments. If there isn't exactly one bean
    of the constructor argument type in the bean factory, a fatal error is raised.

    5. "autodetect"
    Chooses "constructor" or "byType" through introspection of the bean class.
    If a default constructor is found, "byType" gets applied.

    The latter two are similar to PicoContainer and make bean factories simple to
    configure for small namespaces, but doesn't work as well as standard Spring
    behaviour for bigger applications.

    Note that explicit dependencies, i.e. "property" and "constructor-arg" elements,
    always override autowiring. Autowire behaviour can be combined with dependency
```

```
    checking, which will be performed after all autowiring has been completed.
-->
<!ATTLIST bean autowire (no | byName | byType | constructor | autodetect | default) "default">

<!--
    Optional attribute controlling whether to check whether all this
    beans dependencies, expressed in its properties, are satisfied.
    Default is no dependency checking.

    "simple" type dependency checking includes primitives and String
    "object" includes collaborators (other beans in the factory)
    "all" includes both types of dependency checking
-->
<!ATTLIST bean dependency-check (none | objects | simple | all | default) "default">

<!--
    The names of the beans that this bean depends on being initialized.
    The bean factory will guarantee that these beans get initialized before.

    Note that dependencies are normally expressed through bean properties or
    constructor arguments. This property should just be necessary for other kinds
    of dependencies like statics (*ugh*) or database preparation on startup.
-->
<!ATTLIST bean depends-on CDATA #IMPLIED>

<!--
    Optional attribute for the name of the custom initialization method
    to invoke after setting bean properties. The method must have no arguments,
    but may throw any exception.
-->
<!ATTLIST bean init-method CDATA #IMPLIED>

<!--
    Optional attribute for the name of the custom destroy method to invoke
    on bean factory shutdown. The method must have no arguments,
    but may throw any exception. Note: Only invoked on singleton beans!
-->
<!ATTLIST bean destroy-method CDATA #IMPLIED>

<!--
    Bean definitions can specify zero or more constructor arguments.
    They correspond to either a specific index of the constructor argument list
    or are supposed to be matched generically by type.
    This is an alternative to "autowire constructor".
-->
<!ELEMENT constructor-arg (
        description?,
        (bean | ref | idref | list | set | map | props | value | null)
)>

<!--
    The constructor-arg tag can have an optional index attribute,
    to specify the exact index in the constructor argument list. Only needed
    to avoid ambiguities, e.g. in case of 2 arguments of the same type.
-->
<!ATTLIST constructor-arg index CDATA #IMPLIED>

<!--
    The constructor-arg tag can have an optional type attribute,
    to specify the exact type of the constructor argument. Only needed
    to avoid ambiguities, e.g. in case of 2 single argument constructors
    that can both be converted from a String.
-->
<!ATTLIST constructor-arg type CDATA #IMPLIED>

<!--
    Bean definitions can have zero or more properties.
    Property elements correspond to JavaBean setter methods exposed
    by the bean classes. Spring supports primitives, references to other
    beans in the same or related factories, lists, maps and properties.
-->
<!ELEMENT property (
        description?,
        (bean | ref | idref | list | set | map | props | value | null)
)>

<!--
```

```
    The property name attribute is the name of the JavaBean property.
    This follows JavaBean conventions: a name of "age" would correspond
    to setAge()/optional getAge() methods.
-->
<!ATTLIST property name CDATA #REQUIRED>

<!--
    Defines a reference to another bean in this factory or an external
    factory (parent or included factory).
-->
<!ELEMENT ref EMPTY>

<!--
    References must specify a name of the target bean.
    The "bean" attribute can reference any name from any bean in the context,
    to be checked at runtime.
    Local references, using the "local" attribute, have to use bean ids;
    they can be checked by this DTD, thus should be preferred for references
    within the same bean factory XML file.
-->
<!ATTLIST ref bean CDATA #IMPLIED>
<!ATTLIST ref local IDREF #IMPLIED>

<!--
    Defines a string property value, which must also be the id of another
    bean in this factory or an external factory (parent or included factory).
    While a regular 'value' element could instead be used for the same effect,
    using idref in this case allows validation of local bean ids by the xml
    parser, and name completion by helper tools.
-->
<!ELEMENT idref EMPTY>

<!--
    ID refs must specify a name of the target bean.
    The "bean" attribute can reference any name from any bean in the context,
    to be checked at runtime by future Spring implementations.
    Local references, using the "local" attribute, have to use bean ids;
    they can be checked by this DTD, thus should be preferred for references
    within the same bean factory XML file.
-->
<!ATTLIST idref bean CDATA #IMPLIED>
<!ATTLIST idref local IDREF #IMPLIED>

<!--
    A list can contain multiple inner bean, ref, collection, or value elements.
    Java lists are untyped, pending generics support in Java 1.5,
    although references will be strongly typed.
    A list can also map to an array type. The necessary conversion
    is automatically performed by AbstractBeanFactory.
-->
<!ELEMENT list (
        (bean | ref | idref | list | set | map | props | value | null)*
)>

<!--
    A set can contain multiple inner bean, ref, collection, or value elements.
    Java sets are untyped, pending generics support in Java 1.5,
    although references will be strongly typed.
-->
<!ELEMENT set (
        (bean | ref | idref | list | set | map | props | value | null)*
)>

<!--
    A Spring map is a mapping from a string key to object.
    Maps may be empty.
-->
<!ELEMENT map (
        (entry)*
)>

<!--
    A map entry can be an inner bean, ref, collection, or value.
    The name of the property is given by the "key" attribute.
-->
<!ELEMENT entry (
        (bean | ref | idref | list | set | map | props | value | null)
```

```
)>

<!ATTLIST entry key CDATA #REQUIRED>

<!--
    Props elements differ from map elements in that values must be strings.
    Props may be empty.
-->
<!ELEMENT props (
        (prop)*
)>

<!--
    Element content is the string value of the property. The "key"
    attribute is the name of the property.
-->
<!ELEMENT prop
        (#PCDATA)
>

<!--
    Each property element must specify the key. The value is the
    element content: Note that whitespace is trimmed off to avoid
    unwanted whitespace caused by typical XML formatting.
-->
<!ATTLIST prop key CDATA #REQUIRED>

<!--
    Contains a string representation of a property value.
    The property may be a string, or may be converted to the
    required type using the JavaBeans PropertyEditor
    machinery. This makes it possible for application developers
    to write custom PropertyEditor implementations that can
    convert strings to objects.

    Note that this is recommended for simple objects only.
    Configure more complex objects by setting JavaBean
    properties to references to other beans.
-->
<!ELEMENT value (#PCDATA)>

<!--
    Denotes a Java null value. Necessary because an empty "value" tag
    will resolve to an empty String, which will not be resolved to a
    null value unless a special PropertyEditor does so.
-->
<!ELEMENT null (#PCDATA)>
```