java/j2ee Application Framework

2.0.8

Preface

Developing software applications is hard enough even with good tools and technologies. Implementing applications using platforms which promise everything but turn out to be heavy-weight, hard to control and not very efficient during the development cycle makes it even harder. Spring provides a light-weight solution for building enterprise-ready applications, while still supporting the possibility of using declarative transaction management, remote access to your logic using RMI or web services, and various options for persisting your data to a database. Spring provides a full-featured MVC framework, and transparent ways of integrating AOP into your software.

Spring could potentially be a one-stop-shop for all your enterprise applications; however, Spring is modular, allowing you to use just those parts of it that you need, without having to bring in the rest. You can use the IoC container, with Struts on top, but you could also choose to use just the Hibernate integration code or the JDBC abstraction layer. Spring has been (and continues to be) designed to be non-intrusive, meaning dependencies on the framework itself are generally none (or absolutely minimal, depending on the area of use).

This document provides a reference guide to Spring's features. Since this document is still to be considered very much work-in-progress, if you have any requests or comments, please post them on the user mailing list or on the support forums at http://forum.springframework.org/.

Before we go on, a few words of gratitude are due to Christian Bauer (of the Hibernate team), who prepared and adapted the DocBook-XSL software in order to be able to create Hibernate's reference guide, thus also allowing us to create this one. Also thanks to Russell Healy for doing an extensive and valuable review of some of the material.

Chapter 1. Introduction

Background

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?*". Fowler then suggested renaming the principle (or at least giving it a more self-explanatory name), and started to use the term *Dependency Injection*. His article then continued to explain the ideas underpinning the Inversion of Control (IoC) and Dependency Injection (DI) principle.

If you need a decent insight into IoC and DI, please do refer to said article : http://martinfowler.com/articles/injection.html.

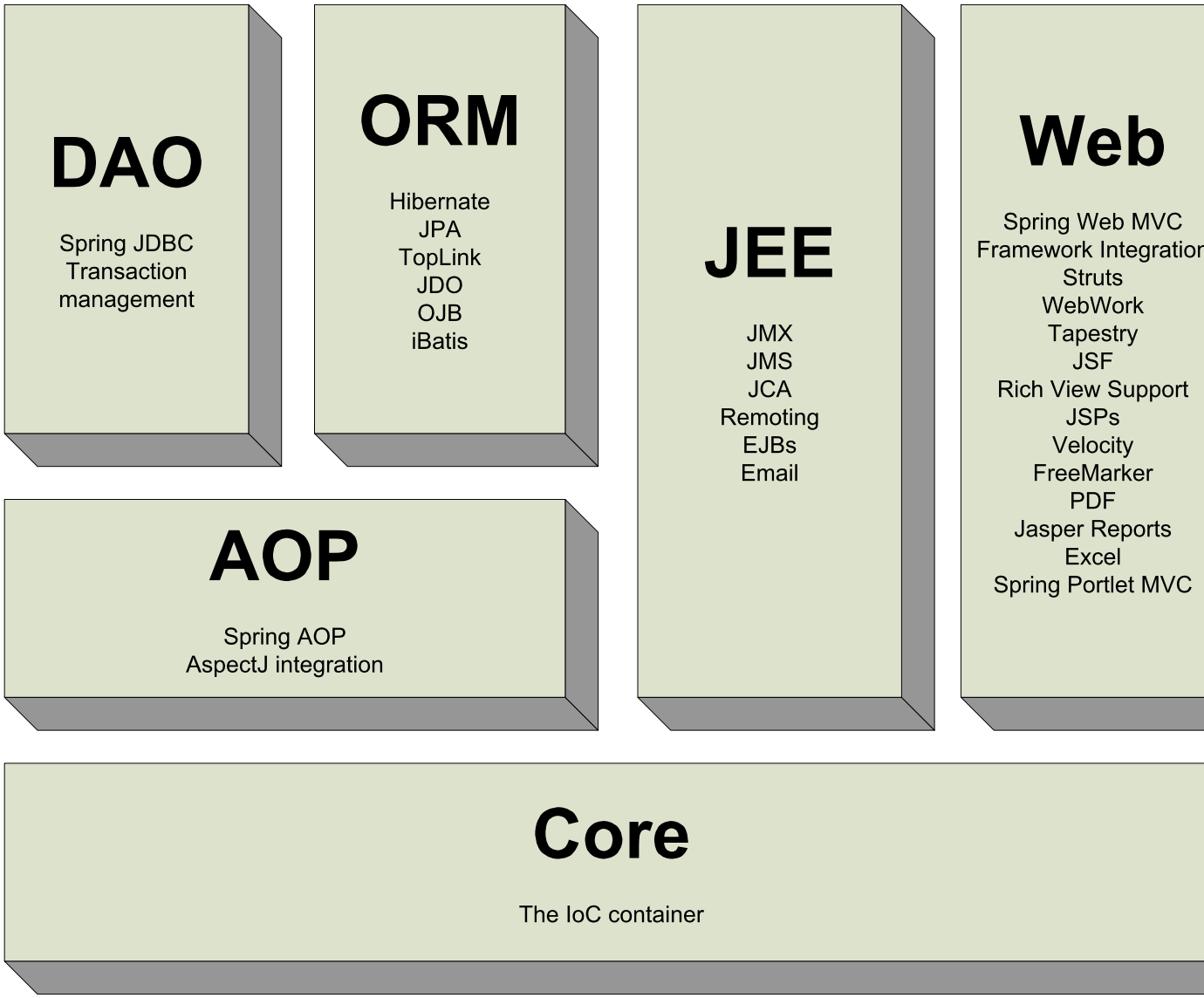
Java applications (a loose term which runs the gamut from constrained applets to full-fledged n-tier server-side enterprise applications) typically are composed of a number of objects that collaborate with one another to form the application proper. The objects in an application can thus be said to have *dependencies* between themselves.

The Java language and platform provides a wealth of functionality for architecting and building applications, ranging all the way from the very basic building blocks of primitive types and classes (and the means to define new classes), to rich full-featured application servers and web frameworks. One area that is decidedly conspicuous by its absence is any means of taking the basic building blocks and composing them into a coherent whole; this area has typically been left to the purvey of the architects and developers tasked with building an application (or applications). Now to be fair, there are a number of design patterns devoted to the business of composing the various classes and object instances that makeup an all-singing, all-dancing application. Design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* (to name but a few) have widespread recognition and acceptance within the software development industry (presumably that is why these patterns have been formalized as patterns in the first place). This is all very well, but these patterns are just that: best practices given a name, typically together with a description of what the pattern does, where the pattern is typically best applied, the problems that the application of the pattern addresses, and so forth. Notice that the last paragraph used the phrase "... a *description* of what the pattern does..."; pattern books and wikis are typically listings of such formalized best practice that you can certainly take away, mull over, and then *implement yourself* in your application.

The IoC component of the Spring Framework addresses the enterprise concern of taking the classes, objects, and services that are to compose an application, by providing a formalized means of composing these various disparate components into a fully working application ready for use. The Spring Framework takes best practices that have been proven over the years in numerous applications and formalized as design patterns, and actually codifies these patterns as first class objects that you as an architect and developer can take away and integrate into your own application(s). This is a *Very Good Thing Indeed* as attested to by the numerous organizations and institutions that have used the Spring Framework to engineer robust, *maintainable* applications.

1.1. Overview

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



Overview of the Spring Framework

The *Core* package is the most fundamental part of the framework and provides the IoC and Dependency

Injection features. The basic concept here is the `BeanFactory`, which provides a sophisticated implementation of the factory pattern which removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The *Context* package build on the solid base provided by the *Core* package: it provides a way to access objects in a framework-style manner in a fashion somewhat reminiscent of a JNDI-registry. The context package inherits its features from the beans package and adds support for internationalization (I18N) (using for example resource bundles), event-propagation, resource-loading, and the 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 of 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 (plain old Java objects)*.

The *ORM* package provides integration layers for popular object-relational mapping APIs, including JPA, 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, such as the simple declarative transaction management feature mentioned previously.

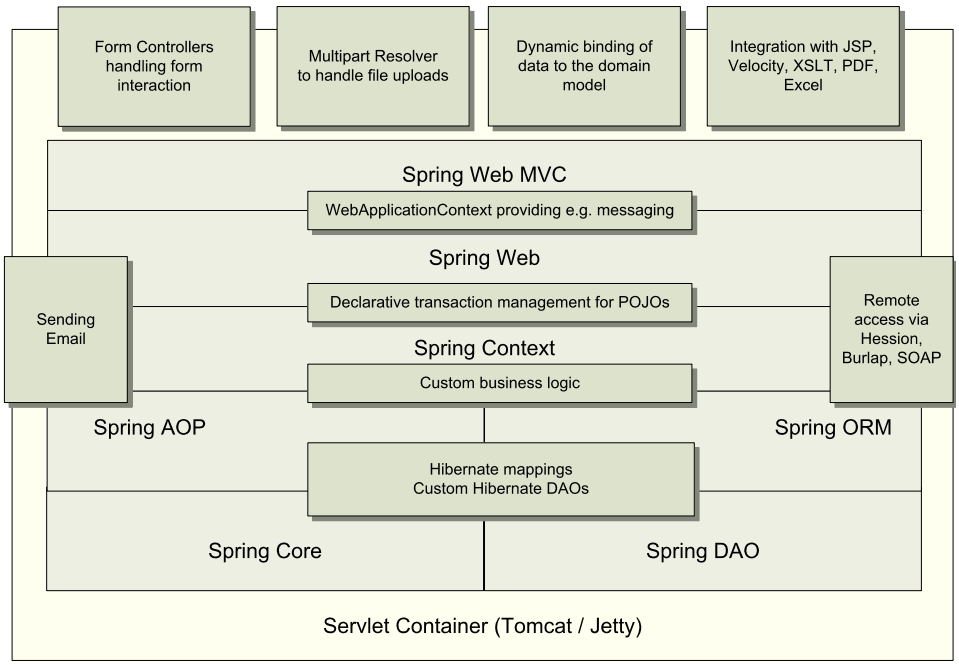
Spring's *AOP* package provides an *AOP Alliance*-compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated. Using source-level metadata functionality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

Spring's *Web* package provides basic web-oriented integration features, such as multipart file-upload functionality, the initialization of the IoC container 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's *MVC* package provides a Model-View-Controller (MVC) implementation for web-applications. Spring's MVC framework is not just any old 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.

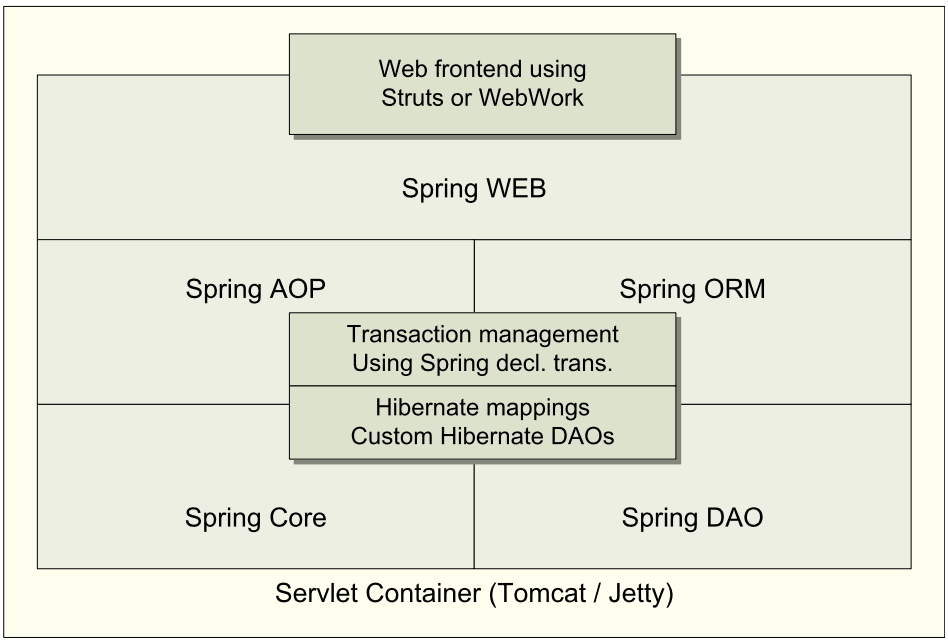
1.2. Usage scenarios

With the building blocks described above you can use Spring in all sorts of scenarios, from applets up to fully-fledged enterprise applications using Spring's transaction management functionality and web framework integration.



Typical full-fledged Spring web application

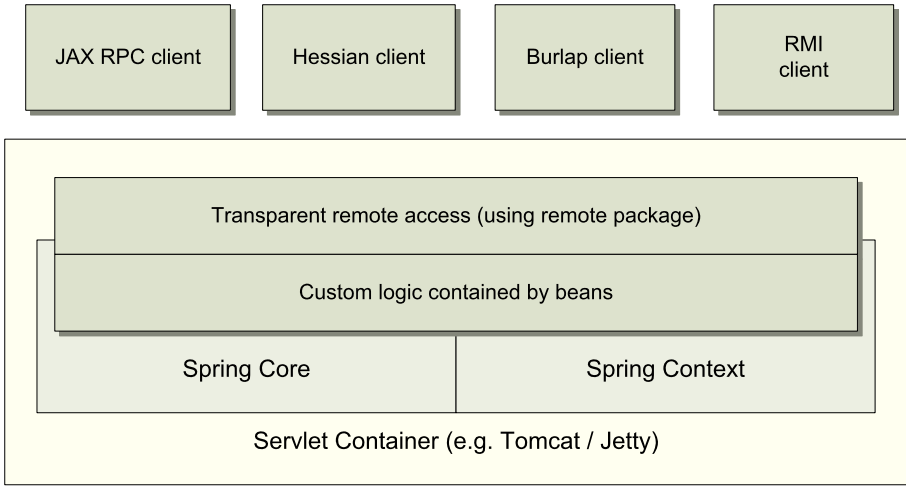
By using Spring's declarative transaction management features the web application is fully transactional, just as it would be when using container managed transactions as provided by Enterprise JavaBeans. All your custom business logic can be implemented using simple POJOs, managed by Spring's IoC container. Additional services include support for sending email, and validation that is independent of the web layer enabling you to choose where to execute validation rules. Spring's ORM support is integrated with JPA, Hibernate, JDO and iBatis; for example, when using Hibernate, you can continue to use your existing mapping files and standard Hibernate `SessionFactory` configuration. 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

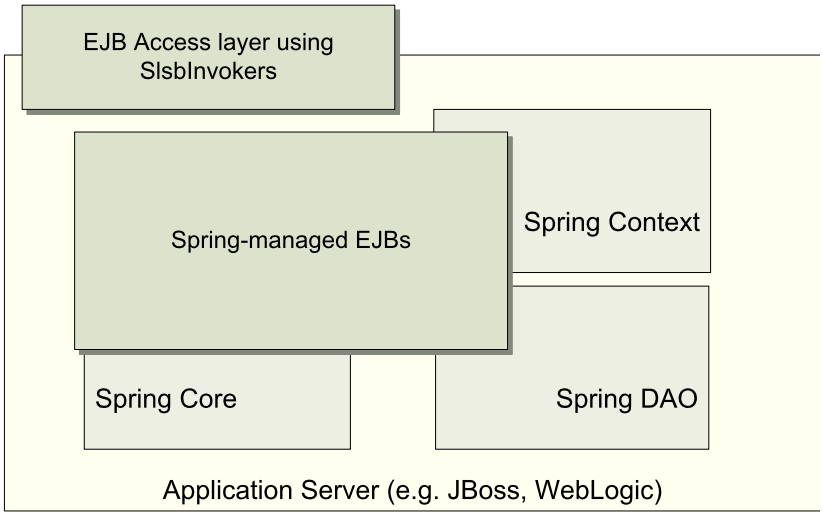
Sometimes the current circumstances do not allow you to completely switch to a different framework. The Spring Framework does *not* force you to use everything within it; it is not an *all-or-nothing* solution. Existing

front-ends built using WebWork, Struts, Tapestry, or other UI frameworks 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 integrate your web layer using a `WebApplicationContext`.



Remoting usage scenario

When you need to access existing code via web services, you can use Spring's `Hessian-`, `Burlap-`, `Rmi-` or `JaxRpcProxyFactory` classes. Enabling remote access to existing applications suddenly is not that hard anymore.



EJBs - Wrapping existing POJOs

The Spring Framework also provides an access- and abstraction- layer for Enterprise JavaBeans, enabling you to reuse your existing POJOs and wrap them in Stateless Session Beans, for use in scalable, failsafe web applications that might need declarative security.

Chapter 2. What's new in Spring 2.0?

2.1. Introduction

If you have been using the Spring Framework for some time, you will be aware that Spring has just undergone a major revision.

> **JDK Support**
>
> The Spring Framework continues to be totally compatible with all versions of Java since (and including) Java 1.3. This means that 1.3, 1.4, and 1.5 are supported, although some advanced functionality of the Spring Framework may not be available to you if you are (for example) committed to using Java 1.3.

This revision includes a host of new features, and a lot of the existing functionality has been reviewed and improved. In fact, so much of Spring is shiny and improved that the Spring development team decided that the next release of Spring merited an increment of the version number; and so Spring 2.0 was announced in December 2005 at the Spring Experience conference in Florida.

This chapter is a guide to the new and improved features of Spring 2.0. It is intended to provide a high-level summary so that seasoned Spring architects and developers can become immediately familiar with the new Spring 2.0 functionality. For more in-depth information on the features, please refer to the corresponding sections hyperlinked from within this chapter.

Some of the new and improved functionality described below has been (or will be) backported into the Spring 1.2.x release line. Please do consult the changelogs for the 1.2.x releases to see if a feature has been backported.

2.2. The Inversion of Control (IoC) container

One of the areas that contains a considerable number of 2.0 improvements is Spring's IoC container.

2.2.1. Easier XML configuration

Spring XML configuration is now even easier, thanks to the advent of the new XML configuration syntax based on XML Schema. If you want to take advantage of the new tags that Spring provides (and the Spring team certainly suggest that you do because they make configuration less verbose and easier to read), then do read the section entitled Appendix A, *XML Schema-based configuration*.

On a related note, there is a new, updated DTD for Spring 2.0 that you may wish to reference if you cannot take advantage of the XML Schema-based configuration. The DOCTYPE declaration is included below for your convenience, but the interested reader should definitely read the `'spring-beans-2.0.dtd'` DTD included in the `'dist/resources'` directory of the Spring 2.0 distribution.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
```

2.2.2. New bean scopes

Previous versions of Spring had IoC container level support for exactly two distinct bean scopes (singleton and prototype). Spring 2.0 improves on this by not only providing a number of additional scopes depending on the environment in which Spring is being deployed (for example, request and session scoped beans in a web environment), but also by providing integration points so that Spring users can create their own scopes.

It should be noted that although the underlying (and internal) implementation for singleton- and prototype-scoped beans has been changed, this change is totally transparent to the end user... no existing configuration needs to change, and no existing configuration will break.

Both the new and the original scopes are detailed in the section entitled Section 3.4, “Bean scopes”.

2.2.3. Extensible XML authoring

Not only is XML configuration easier to write, it is now also extensible.

What 'extensible' means in this context is that you, as an application developer, or (more likely) as a third party framework or product vendor, can write custom tags that other developers can then plug into their own Spring configuration files. This allows you to have your own domain specific language (the term is used loosely here) of sorts be reflected in the specific configuration of your own components.

Implementing custom Spring tags may not be of interest to every single application developer or enterprise architect using Spring in their own projects. We expect third-party vendors to be highly interested in developing custom configuration tags for use in Spring configuration files.

The extensible configuration mechanism is documented in Appendix B, *Extensible XML authoring*.

2.3. Aspect Oriented Programming (AOP)

Spring 2.0 has a much improved AOP offering. The Spring AOP framework itself is markedly easier to configure in XML, and significantly less verbose as a result; and Spring 2.0 integrates with the AspectJ pointcut language and @AspectJ aspect declaration style. The chapter entitled Chapter 6, *Aspect Oriented Programming with Spring* is dedicated to describing this new support.

2.3.1. Easier AOP XML configuration

Spring 2.0 introduces new schema support for defining aspects backed by regular Java objects. This support takes advantage of the AspectJ pointcut language and offers fully typed advice (i.e. no more casting and `Object[]` argument manipulation). Details of this support can be found in the section entitled Section 6.3, “Schema-based AOP support”.

2.3.2. Support for @AspectJ aspects

Spring 2.0 also supports aspects defined using the @AspectJ annotations. These aspects can be shared between AspectJ and Spring AOP, and require (honestly!) only some simple configuration. Said support for @AspectJ aspects is discussed in Section 6.2, “@AspectJ support”.

2.4. The Middle Tier

2.4.1. Easier configuration of declarative transactions in XML

The way that transactions are configured in Spring 2.0 has been changed significantly. The previous 1.2.x style of configuration continues to be valid (and supported), but the new style is markedly less verbose and is the recommended style. Spring 2.0 also ships with an AspectJ aspects library that you can use to make pretty much any object transactional - even objects not created by the Spring IoC container.

The chapter entitled Chapter 9, *Transaction management* contains all of the details.

2.4.2. JPA

Spring 2.0 ships with a JPA abstraction layer that is similar in intent to Spring's JDBC abstraction layer in terms of scope and general usage patterns.

If you are interested in using a JPA-implementation as the backbone of your persistence layer, the section entitled Section 12.6, “JPA” is dedicated to detailing Spring's support and value-add in this area.

2.4.3. Asynchronous JMS

Prior to Spring 2.0, Spring's JMS offering was limited to sending messages and the *synchronous* receiving of messages. This functionality (encapsulated in the `JmsTemplate` class) is great, but it doesn't address the requirement for the *asynchronous* receiving of messages.

Spring 2.0 now ships with full support for the reception of messages in an asynchronous fashion, as detailed in the section entitled Section 19.4.2, “Asynchronous Reception - Message-Driven POJOs”.

2.4.4. JDBC

There are some small (but nevertheless notable) new classes in the Spring Framework's JDBC support library. The first, `NamedParameterJdbcTemplate`, provides support for programming JDBC statements using named parameters (as opposed to programming JDBC statements using only classic placeholder (`'?'`) arguments.

Another of the new classes, the `SimpleJdbcTemplate`, is aimed at making using the `JdbcTemplate` even easier to use when you are developing against Java 5+.

2.5. The Web Tier

The web tier support has been *substantially* improved and expanded in Spring 2.0.

2.5.1. A form tag library for Spring MVC

A rich JSP tag library for Spring MVC was *the* JIRA issue that garnered the most votes from Spring users (by a wide margin).

Spring 2.0 ships with a full featured JSP tag library that makes the job of authoring JSP pages much easier when using Spring MVC; the Spring team is confident it will satisfy all of those developers who voted for the issue on JIRA. The new tag library is itself covered in the section entitled Section 13.9, “Using Spring's form tag library”, and a quick reference to all of the new tags can be found in the appendix entitled Appendix E, *spring-form.tld*.

2.5.2. Sensible defaulting in Spring MVC

For a lot of projects, sticking to established conventions and having reasonable defaults is just what the projects need... this theme of convention-over-configuration now has explicit support in Spring MVC. What this means is that if you establish a set of naming conventions for your `Controllers` and views, you can *substantially* cut down on the amount of XML configuration that is required to setup handler mappings, view resolvers, `ModelAndView` instances, etc. This is a great boon with regards to rapid prototyping, and can also lend a degree of (always good-to-have) consistency across a codebase.

Spring MVC's convention-over-configuration support is detailed in the section entitled Section 13.11, “Convention over configuration”

2.5.3. Portlet framework

Spring 2.0 ships with a Portlet framework that is conceptually similar to the Spring MVC framework. Detailed coverage of the Spring Portlet framework can be found in the section entitled Chapter 16, *Portlet MVC Framework*.

2.6. Everything else

This final section outlines all of the other new and improved Spring 2.0 features and functionality.

2.6.1. Dynamic language support

Spring 2.0 now has support for beans written in languages other than Java, with the currently supported dynamic languages being JRuby, Groovy and BeanShell. This dynamic language support is comprehensively detailed in the section entitled Chapter 24, *Dynamic language support*.

2.6.2. JMX

The Spring Framework now has support for `Notifications`; it is also possible to exercise declarative control over the registration behavior of MBeans with an `MBeanServer`.

- Section 20.7, “Notifications”
- Section 20.2.5, “Controlling the registration behavior”

2.6.3. Task scheduling

Spring 2.0 offers an abstraction around the scheduling of tasks. For the interested developer, the section entitled Section 23.4, “The Spring `TaskExecutor` abstraction” contains all of the details.

2.6.4. Java 5 (Tiger) support

Find below pointers to documentation describing some of the new Java 5 support in Spring 2.0.

- Section 9.5.6, “Using `@Transactional`”

- Section 25.3.1, “`@Required`”
- Section 11.2.3, “`SimpleJdbcTemplate`”
- Section 12.6, “JPA”
- Section 6.2, “@AspectJ support”
- Section 6.8.1, “Using AspectJ to dependency inject domain objects with Spring”

2.7. Migrating to Spring 2.0

This final section details issues that may arise during any migration from Spring 1.2.x to Spring 2.0. Feel free to take this next statement with a pinch of salt, but upgrading to Spring 2.0 from a Spring 1.2 application *should* simply be a matter of dropping the Spring 2.0 jar into the appropriate location in your application's directory structure.

The keyword from the last sentence was of course the “*should*”. Whether the upgrade is seamless or not depends on how much of the Spring APIs you are using in your code. Spring 2.0 removed pretty much all of the classes and methods previously marked as deprecated in the Spring 1.2.x codebase, so if you have been using such classes and methods, you will of course have to use alternative classes and methods (some of which are summarized below).

With regards to configuration, Spring 1.2.x style XML configuration is 100%, satisfaction-guaranteed compatible with the Spring 2.0 library. Of course if you are still using the Spring 1.2.x DTD, then you won't be able to take advantage of some of the new Spring 2.0 functionality (such as scopes and easier AOP and transaction configuration), but nothing will blow up.

The suggested migration strategy is to drop in the Spring 2.0 jar(s) to benefit from the improved code present in the release (bug fixes, optimizations, etc.). You can then, on an incremental basis, choose to start using the new Spring 2.0 features and configuration. For example, you could choose to start configuring just your aspects in the new Spring 2.0 style; it is perfectly valid to have 90% of your configuration using the old-school Spring 1.2.x configuration (which references the 1.2.x DTD), and have the other 10% using the new Spring 2.0 configuration (which references the 2.0 DTD or XSD). Bear in mind that you are not forced to upgrade your XML configuration should you choose to drop in the Spring 2.0 libraries.

2.7.1. Changes

For a comprehensive list of changes, consult the `'changelog.txt'` file that is located in the top level directory of the Spring Framework 2.0 distribution.

2.7.1.1. Jar packaging

The packaging of the Spring Framework jars has changed quite substantially between the 1.2.x and 2.0 releases. In particular, there are now dedicated jars for the JDO, Hibernate 2/3, TopLink ORM integration classes: they are no longer bundled in the core `'spring.jar'` file anymore.

2.7.1.2. XML configuration

Spring 2.0 ships with XSDs that describe Spring's XML metadata format in a much richer fashion than the DTD that shipped with previous versions. The old DTD is still fully supported, but if possible you are encouraged to reference the XSD files at the top of your bean definition files.

One thing that has changed in a (somewhat) breaking fashion is the way that bean scopes are defined. If you are using the Spring 1.2 DTD you can continue to use the `'singleton'` attribute. You can however choose to reference the new Spring 2.0 DTD which does not permit the use of the `'singleton'` attribute, but rather uses the `'scope'` attribute to define the bean lifecycle scope.

2.7.1.3. Deprecated classes and methods

A number of classes and methods that previously were marked as `@deprecated` have been removed from the Spring 2.0 codebase. The Spring team decided that the 2.0 release marked a fresh start of sorts, and that any deprecated 'cruft' was better excised now instead of continuing to haunt the codebase for the foreseeable future.

As mentioned previously, for a comprehensive list of changes, consult the `'changelog.txt'` file that is located in the top level directory of the Spring Framework 2.0 distribution.

The following classes/interfaces have been removed from the Spring 2.0 codebase:

- `ResultReader` : Use the `RowMapper` interface instead.
- `BeanFactoryBootstrap` : Consider using a `BeanFactoryLocator` or a custom bootstrap class instead.

2.7.1.4. Apache OJB

Please note that support for Apache OJB was *totally removed* from the main Spring source tree; the Apache OJB integration library is still available, but can be found in it's new home in the Spring Modules project.

2.7.1.5. iBatis

Please note that support for iBATIS SQL Maps 1.3 has been removed. If you haven't done so already, upgrade to iBATIS SQL Maps 2.0/2.1.

2.7.1.6. `UrlFilenameViewController`

The view name that is determined by the `UrlFilenameViewController` now takes into account the nested path of the request. This is a breaking change from the original contract of the `UrlFilenameViewController`, and means that if you are upgrading to Spring 2.0 from Spring 1.x and you are using this class you *might* have to change your Spring Web MVC configuration slightly. Refer to the class level Javadocs of the `UrlFilenameViewController` to see examples of the new contract for view name determination.

2.8. Updated sample applications

A number of the sample applications have also been updated to showcase the new and improved features of Spring 2.0, so do take the time to investigate them. The aforementioned sample applications can be found in the `'samples'` directory of the full Spring distribution (`'spring-with-dependencies.[zip|tar.gz]'`), and are documented (in part) in the chapter entitled Chapter 26, *Showcase applications*.

2.9. Improved documentation

The Spring reference documentation has also substantially been updated to reflect all of the above features new in Spring 2.0. While every effort has been made to ensure that there are no errors in this documentation, some errors may nevertheless have crept in. If you do spot any typos or even more serious errors, and you can spare a

few cycles during lunch, please do bring the error to the attention of the Spring team by raising an issue.

Special thanks to Arthur Loder for his tireless proofreading of the Spring Framework reference documentation and Javadocs.

Part I. Core Technologies

This initial part of the reference documentation covers all of those technologies that are absolutely integral to the Spring Framework.

Foremost amongst these is the Spring Framework's Inversion of Control (IoC) container. A thorough treatment of the Spring Framework's IoC container is closely followed by comprehensive coverage of Spring's Aspect-Oriented Programming (AOP) technologies. The Spring Framework has its own AOP framework, which is conceptually easy to understand, and which successfully addresses the 80% sweet spot of AOP requirements in Java enterprise programming.

Coverage of Spring's integration with AspectJ (currently the richest - in terms of features - and certainly most mature AOP implementation in the Java enterprise space) is also provided.

Finally, the adoption of the test-driven-development (TDD) approach to software development is certainly advocated by the Spring team, and so coverage of Spring's support for integration testing is covered (alongside best practices for unit testing). The Spring team have found that the correct use of IoC certainly does make both unit and integration testing easier (in that the presence of setter methods and appropriate constructors on classes makes them easier to wire together on a test without having to set up service locator registries and suchlike)... the chapter dedicated solely to testing will hopefully convince you of this as well.

- Chapter 3, *The IoC container*
- Chapter 4, *Resources*
- Chapter 5, *Validation, Data-binding, the* `BeanWrapper`*, and* `PropertyEditors`
- Chapter 6, *Aspect Oriented Programming with Spring*
- Chapter 7, *Spring AOP APIs*
- Chapter 8, *Testing*

Chapter 3. The IoC container

3.1. Introduction

This chapter covers the Spring Framework's implementation of the Inversion of Control (IoC) [1] principle.

> **`BeanFactory` or `ApplicationContext`?**
>
> Users are sometimes unsure whether a `BeanFactory` or an `ApplicationContext` is best suited for use in a particular situation. A `BeanFactory` pretty much just instantiates and configures beans. An `ApplicationContext` also does that, *and* it provides the supporting infrastructure to enable *lots* of enterprise-specific features such as transactions and AOP.
>
> *In short, favor the use of an `ApplicationContext`.*

The `org.springframework.beans` and `org.springframework.context` packages provide the basis for the Spring Framework's IoC container. The `BeanFactory` interface provides an advanced configuration mechanism capable of managing objects of any nature. The `ApplicationContext` interface builds on top of the `BeanFactory` (it is a sub-interface) and adds other functionality such as easier integration with Spring's AOP features, message resource handling (for use in internationalization), event propagation, and application-layer specific contexts such as the `WebApplicationContext` for use in web applications.

In short, the `BeanFactory` provides the configuration framework and basic functionality, while the `ApplicationContext` adds more enterprise-centric functionality to it. The `ApplicationContext` is a complete superset of the `BeanFactory`, and any description of `BeanFactory` capabilities and behavior is to be considered to apply to the `ApplicationContext` as well.

This chapter is divided into two parts, with the first part covering the basic principles that apply to both the `BeanFactory` and `ApplicationContext`, and with the second part covering those features that apply only to the `ApplicationContext` interface.

3.2. Basics - containers and beans

In Spring, those objects that form the backbone of your application and that are managed by the Spring IoC *container* are referred to as *beans*. A bean is simply an object that is instantiated, assembled and otherwise managed by a Spring IoC container; other than that, there is nothing special about a bean (it is in all other respects one of probably many objects in your application). These beans, and the *dependencies* between them, are reflected in the *configuration metadata* used by a container.

> **Why... *bean*?**
>
> The motivation for using the name *'bean'*, as opposed to *'component'* or *'object'* is rooted in the origins of the Spring Framework itself (it arose partly as a response to the complexity of Enterprise Java*Beans*).

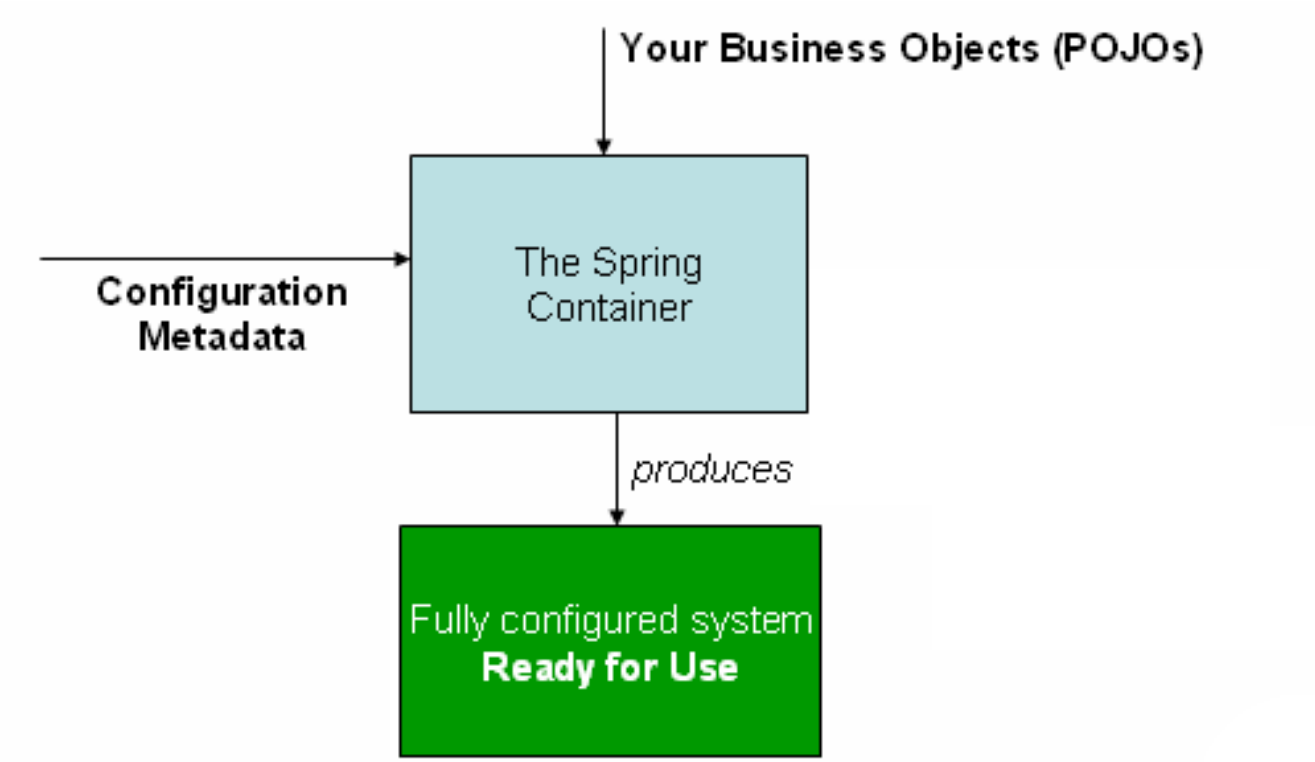
3.2.1. The container

[1] See the section entitled Background

The `org.springframework.beans.factory.BeanFactory` is the actual representation of the Spring IoC *container* that is responsible for containing and otherwise managing the aforementioned beans.

The `BeanFactory` interface is the central IoC container interface in Spring. Its responsibilities include instantiating or sourcing application objects, configuring such objects, and assembling the dependencies between these objects.

There are a number of implementations of the `BeanFactory` interface that come supplied straight out-of-the-box with Spring. The most commonly used `BeanFactory` implementation is the `XmlBeanFactory` class. This implementation allows you to express the objects that compose your application, and the doubtless rich interdependencies between such objects, in terms of XML. The `XmlBeanFactory` takes this XML *configuration metadata* and uses it to create a fully configured system or application.



The Spring IoC container

3.2.1.1. Configuration metadata

As can be seen in the above image, the Spring IoC container consumes some form of *configuration metadata*; this configuration metadata is nothing more than how you (as an application developer) inform the Spring container as to how to "*instantiate, configure, and assemble [the objects in your application]*". This configuration metadata is typically supplied in a simple and intuitive XML format. When using XML-based configuration metadata, you write *bean definitions* for those beans that you want the Spring IoC container to manage, and then let the container do it's stuff.

Note

XML-based metadata is by far the most commonly used form of configuration metadata. It is *not* however the only form of configuration metadata that is allowed. The Spring IoC container itself is *totally* decoupled from the format in which this configuration metadata is actually written. At the time of writing, you can supply this configuration metadata using either XML, the Java properties format, or programmatically (using Spring's public API). The XML-based configuration metadata

format really is simple though, and so the remainder of this chapter will use the XML format to convey key concepts and features of the Spring IoC container.

Resources

Once you have learned the basics of the IoC container (this chapter), it will also be useful to learn about Spring's `Resource` abstraction, as described in Chapter 4, *Resources*.

The location path or paths supplied to an `ApplicationContext` constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java `CLASSPATH`, etc.

Please be advised that in the vast majority of application scenarios, explicit user code is not required to instantiate one or more instances of a Spring IoC container. For example, in a web application scenario, a simple eight (or so) lines of absolutely boilerplate J2EE web descriptor XML in the `web.xml` file of the application will typically suffice (see Section 3.8.4, “Convenient `ApplicationContext` instantiation for web applications”).

Spring configuration consists of at least one bean definition that the container must manage, but typically there will be more than one bean definition. When using XML-based configuration metadata, these beans are configured as `<bean/>` elements inside a top-level `<beans/>` element.

These bean definitions correspond to the actual objects that make up your application. Typically you will have bean definitions for your service layer objects, your data access objects (DAOs), presentation objects such as Struts `Action` instances, infrastructure objects such as Hibernate `SessionFactory` instances, JMS `Queue` references, etc. (the possibilities are of course endless, and are limited only by the scope and complexity of your application). (Typically one does not configure fine-grained domain objects in the container.)

Find below an example of the basic structure of XML-based configuration metadata.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/bea

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <bean id="..." class="...">
    <!-- collaborators and configuration for this bean go here -->
  </bean>

  <!-- more bean definitions go here... -->

</beans>
```

3.2.2. Instantiating a container

Instantiating a Spring IoC container is easy; find below some examples of how to do just that:

```
Resource resource = new FileSystemResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

... or...

```
ClassPathResource resource = new ClassPathResource("beans.xml");
BeanFactory factory = new XmlBeanFactory(resource);
```

... or...

```
ApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml", "applicationContext-part2.xml"});

// of course, an ApplicationContext is just a BeanFactory
BeanFactory factory = context;
```

3.2.2.1. Composing XML-based configuration metadata

It can often be useful to split up container definitions into multiple XML files. One way to then load an application context which is configured from all these XML fragments is to use the application context constructor which takes multiple `Resource` locations. With a bean factory, a bean definition reader can be used multiple times to read definitions from each file in turn.

Generally, the Spring team prefers the above approach, since it keeps container configuration files unaware of the fact that they are being combined with others. An alternate approach is to use one or more occurrences of the `<import/>` element to load bean definitions from another file (or files). Let's look at a sample:

```
<beans>

    <import resource="services.xml"/>
    <import resource="resources/messageSource.xml"/>
    <import resource="/resources/themeSource.xml"/>

    <bean id="bean1" class="..."/>
    <bean id="bean2" class="..."/>

</beans>
```

In this example, external bean definitions are being loaded from 3 files, `services.xml`, `messageSource.xml`, and `themeSource.xml`. All location paths are considered relative to the definition file doing the importing, so `services.xml` in this case must be in the same directory or classpath location as the file doing the importing, while `messageSource.xml` and `themeSource.xml` must be in a `resources` location below the location of the importing file. As you can see, a leading slash is actually ignored, but given that these are considered relative paths, it is probably better form not to use the slash at all.

The contents of the files being imported must be fully valid XML bean definition files according to the Schema or DTD, including the top level `<beans/>` element.

3.2.3. The beans

As mentioned previously, a Spring IoC container manages one or more *beans*. These beans are created using the instructions defined in the configuration metadata that has been supplied to the container (typically in the form of XML `<bean/>` definitions).

Within the container itself, these bean definitions are represented as `BeanDefinition` objects, which contain (among other information) the following metadata:

- *a package-qualified class name:* this is normally the actual implementation class of the bean being defined. However, if the bean is to be instantiated by invoking a `static` *factory* method instead of using a normal constructor, this will actually be the class name of the factory class.

- bean behavioral configuration elements, which state how the bean should behave in the container (prototype or singleton, autowiring mode, initialization and destruction callbacks, and so forth).
- constructor arguments and property values to set in the newly created bean. An example would be the number of connections to use in a bean that manages a connection pool (either specified as a property or as a constructor argument), or the pool size limit.
- other beans which are needed for the bean to do its work, that is *collaborators* (also called dependencies).

The concepts listed above directly translate to a set of properties that each bean definition consists of. Some of these properties are listed below, along with a link to further documentation about each of them.

Table 3.1. The bean definition

Feature	**Explained in...**
class	Section 3.2.3.2, “Instantiating beans”
name	Section 3.2.3.1, “Naming beans”
scope	Section 3.4, “Bean scopes”
constructor arguments	Section 3.3.1, “Injecting dependencies”
properties	Section 3.3.1, “Injecting dependencies”
autowiring mode	Section 3.3.6, “Autowiring collaborators”
dependency checking mode	Section 3.3.7, “Checking for dependencies”
lazy-initialization mode	Section 3.3.5, “Lazily-instantiated beans”
initialization method	Section 3.5.1, “Lifecycle interfaces”
destruction method	Section 3.5.1, “Lifecycle interfaces”

Besides bean definitions which contain information on how to create a specific bean, certain `BeanFactory` implementations also permit the registration of existing objects that have been created outside the factory (by user code). The `DefaultListableBeanFactory` class supports this through the `registerSingleton(..)` method. Typical applications solely work with beans defined through metadata bean definitions, though.

3.2.3.1. Naming beans

Bean naming conventions

The convention (at least amongst the Spring development team) is to use the standard Java convention for instance field names when naming beans. That is, bean names start with a lowercase letter, and are

camel-cased from then on. Examples of such names would be (without quotes) `'accountManager'`, `'accountService'`, `'userDao'`, `'loginController'`, etc.

Adopting a consistent way of naming your beans will go a long way towards making your configuration easier to read and understand; adopting such naming standards is not hard to do, and if you are using Spring AOP it can pay off handsomely when it comes to applying advice to a set of beans related by name.

Every bean has one or more ids (also called identifiers, or names; these terms refer to the same thing). These ids must be unique within the container the bean is hosted in. A bean will almost always have only one id, but if a bean has more than one id, the extra ones can essentially be considered aliases.

When using XML-based configuration metadata, you use the `'id'` or `'name'` attributes to specify the bean identifier(s). The `'id'` attribute allows you to specify exactly one id, and as it is a real XML element ID attribute, the XML parser is able to do some extra validation when other elements reference the id; as such, it is the preferred way to specify a bean id. However, the XML specification does limit the characters which are legal in XML IDs. This is usually not a constraint, but if you have a need to use one of these special XML characters, or want to introduce other aliases to the bean, you may also or instead specify one or more bean ids, separated by a comma (`,`), semicolon (`;`), or whitespace in the `'name'` attribute.

Please note that you are not required to supply a name for a bean. If no name is supplied explicitly, the container will generate a (unique) name for that bean. The motivations for not supplying a name for a bean will be discussed later (one use case is inner beans).

3.2.3.1.1. Aliasing beans

In a bean definition itself, you may supply more than one name for the bean, by using a combination of up to one name specified via the `id` attribute, and any number of other names via the `name` attribute. All these names can be considered equivalent aliases to the same bean, and are useful for some situations, such as allowing each component used in an application to refer to a common dependency using a bean name that is specific to that component itself.

Having to specify all aliases when the bean is actually defined is not always adequate however. It is sometimes desirable to introduce an alias for a bean which is defined elsewhere. In XML-based configuration metadata this may be accomplished via the use of the standalone `<alias/>` element. For example:

```
<alias name="fromName" alias="toName"/>
```

In this case, a bean in the same container which is named `'fromName'`, may also after the use of this alias definition, be referred to as `'toName'`.

As a concrete example, consider the case where component A defines a DataSource bean called componentA-dataSource, in its XML fragment. Component B would however like to refer to the DataSource as componentB-dataSource in its XML fragment. And the main application, MyApp, defines its own XML fragment and assembles the final application context from all three fragments, and would like to refer to the DataSource as myApp-dataSource. This scenario can be easily handled by adding to the MyApp XML fragment the following standalone aliases:

```
<alias name="componentA-dataSource" alias="componentB-dataSource"/>
<alias name="componentA-dataSource" alias="myApp-dataSource" />
```

Now each component and the main app can refer to the dataSource via a name that is unique and guaranteed

not to clash with any other definition (effectively there is a namespace), yet they refer to the same bean.

3.2.3.2. Instantiating beans

Inner class names

If for whatever reason you want to configure a bean definition for a `static` inner class, you have to use the *binary* name of the inner class.

For example, if you have a class called `Foo` in the `com.example` package, and this `Foo` class has a `static` inner class called `Bar`, the value of the `'class'` attribute on a bean definition would be...

```
com.example.Foo$Bar
```

Notice the use of the `$` character in the name to separate the inner class name from the outer class name.

A bean definition can be seen as a recipe for creating one or more actual objects. The container looks at the recipe for a named bean when asked, and uses the configuration metadata encapsulated by that bean definition to create (or acquire) an actual object.

If you are using XML-based configuration metadata, you can specify the type (or class) of object that is to be instantiated using the `'class'` attribute of the `<bean/>` element. This `'class'` attribute (which internally eventually boils down to being a `Class` property on a `BeanDefinition` instance) is normally mandatory (see Section 3.2.3.2.3, “Instantiation using an instance factory method” and Section 3.6, “Bean definition inheritance” for the two exceptions) and is used for one of two purposes. The class property specifies the class of the bean to be constructed in the much more common case where the container itself directly creates the bean by calling its constructor reflectively (somewhat equivalent to Java code using the *'new'* operator). In the less common case where the container invokes a `static`, *factory* method on a class to create the bean, the class property specifies the actual class containing the `static` factory method that is to be invoked to create the object (the type of the object returned from the invocation of the `static` factory method may be the same class or another class entirely, it doesn't matter).

3.2.3.2.1. Instantiation using a constructor

When creating a bean using the constructor approach, all normal classes are usable by and compatible with Spring. That is, the class being created does not need to implement any specific interfaces or be coded in a specific fashion. Just specifying the bean class should be enough. However, depending on what type of IoC you are going to use for that specific bean, you may need a default (empty) constructor.

Additionally, the Spring IoC container isn't limited to just managing true JavaBeans, it is also able to manage virtually *any* class you want it to manage. Most people using Spring prefer to have actual JavaBeans (having just a default (no-argument) constructor and appropriate setters and getters modeled after the properties) in the container, but it is also possible to have more exotic non-bean-style classes in your container. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, Spring can manage it as well.

When using XML-based configuration metadata you can specify your bean class like so:

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

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

The mechanism for supplying arguments to the constructor (if required), or setting properties of the object

instance after it has been constructed, will be described shortly.

3.2.3.2.2. Instantiation using a `static` factory method

When defining a bean which is to be created using a static factory method, along with the `class` attribute which specifies the class containing the `static` factory method, another attribute named `factory-method` is needed to specify the name of the factory method itself. Spring expects to be able to call this method (with an optional list of arguments as described later) and get back a live object, which from that point on is treated as if it had been created normally via a constructor. One use for such a bean definition is to call `static` factories in legacy code.

The following example shows a bean definition which specifies that the bean is to be created by calling a factory-method. Note that the definition does not specify the type (class) of the returned object, only the class containing the factory method. In this example, the `createInstance()` method must be a *static* method.

```
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance"/>
```

The mechanism for supplying (optional) arguments to the factory method, or setting properties of the object instance after it has been returned from the factory, will be described shortly.

3.2.3.2.3. Instantiation using an instance factory method

In a fashion similar to instantiation via a static factory method, instantiation using an instance factory method is where the factory method of an existing bean from the container is invoked to create the new bean.

To use this mechanism, the `'class'` attribute must be left empty, and the `'factory-bean'` attribute must specify the name of a bean in the current (or parent/ancestor) container that contains the factory method. The factory method itself must still be set via the `'factory-method'` attribute (as seen in the example below).

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="myFactoryBean" class="...">
  ...
</bean>

<!-- the bean to be created via the factory bean -->
<bean id="exampleBean"
      factory-bean="myFactoryBean"
      factory-method="createInstance"/>
```

Although the mechanisms for setting bean properties are still to be discussed, one implication of this approach is that the factory bean itself can be managed and configured via DI.

3.2.4. Using the container

A `BeanFactory` is essentially nothing more than the interface for an advanced factory capable of maintaining a registry of different beans and their dependencies. The `BeanFactory` enables you to read bean definitions and access them using the bean factory. When using just the `BeanFactory` you would create one and read in some bean definitions in the XML format as follows:

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

Basically that's all there is to it. Using `getBean(String)` you can retrieve instances of your beans; the client-side view of the `BeanFactory` is surprisingly simple. The `BeanFactory` interface has only six methods for

client code to call:

- `boolean containsBean(String)`: returns true if the `BeanFactory` contains a bean definition or bean instance that matches the given name
- `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, either a singleton and thus shared instance or a newly created bean will be returned. A `BeansException` will be thrown when either the bean could not be found (in which case it'll be a `NoSuchBeanDefinitionException`), or an exception occurred while instantiating 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)
- `Class getType(String name)`: returns the `Class` of the bean with the given name. If no bean corresponding to the given name could be found, a `NoSuchBeanDefinitionException` will be thrown
- `boolean isSingleton(String)`: determines whether or not the bean definition or bean instance registered under the given name is a singleton (bean scopes such as singleton are explained later). If no bean corresponding to the given name could be found, a `NoSuchBeanDefinitionException` will be thrown
- `String[] getAliases(String)`: Return the aliases for the given bean name, if any were defined in the bean definition

3.3. Dependencies

Your typical enterprise application is not made up of a single object (or bean in the Spring parlance). Even the simplest of applications will no doubt have at least a handful of objects that work together to present what the end-user sees as a coherent application. This next section explains how you go from defining a number of bean definitions that stand-alone, each to themselves, to a fully realized application where objects work (or collaborate) together to achieve some goal (usually an application that does what the end-user wants).

3.3.1. Injecting dependencies

The basic principle behind *Dependency Injection* (DI) is that objects define their dependencies (that is to say the other objects they work with) only through constructor arguments, arguments to a factory method, or properties which are set on the object instance after it has been constructed or returned from a factory method. Then, it is the job of the container to actually *inject* those dependencies when it creates the bean. This is fundamentally the inverse, hence the name *Inversion of Control* (IoC), of the bean itself being in control of instantiating or locating its dependencies on its own using direct construction of classes, or something like the *Service Locator* pattern.

It becomes evident upon usage that code gets much cleaner when the DI principle is applied, and reaching a higher grade of decoupling is much easier when beans do not look up their dependencies, but are provided with them (and additionally do not even know where the dependencies are located and of what actual class they are).

As touched on in the previous paragraph, DI exists in two major variants, namely Setter Injection, and Constructor Injection.

3.3.1.1. Setter Injection

Setter-based DI is realized by calling setter methods on your beans after invoking a no-argument constructor or no-argument `static` factory method to instantiate your bean.

Find below an example of a class that can only be dependency injected using pure setter injection. Note that there is nothing *special* about this class... it is plain old Java.

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can 'inject' a MovieFinder
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

3.3.1.2. Constructor Injection

Constructor-based DI is realized by invoking a constructor with a number of arguments, each representing a collaborator. Additionally, calling a `static` factory method with specific arguments to construct the bean, can be considered almost equivalent, and the rest of this text will consider arguments to a constructor and arguments to a `static` factory method similarly.

Find below an example of a class that could only be dependency injected using constructor injection. Again, note that there is nothing *special* about this class.

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a constructor so that the Spring container can 'inject' a MovieFinder
    public SimpleMovieLister(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

Constructor- or Setter-based DI?

The Spring team generally advocates the usage of setter injection, since a large number of constructor arguments can get unwieldy, especially when some properties are optional. The presence of setter methods also makes objects of that class amenable to being re-configured (or re-injected) at some later time (for management via JMX MBeans is a particularly compelling use case).

Constructor-injection is favored by some purists though (and with good reason). Supplying all of an object's dependencies means that that object is never returned to client (calling) code in a less than totally initialized state. The flipside is that the object becomes less amenable to re-configuration (or re-injection).

There is no hard and fast rule here. Use whatever type of DI makes the most sense for a particular class; sometimes, when dealing with third party classes to which you do not have the source, the choice will already have been made for you - a legacy class may not expose any setter methods, and so constructor injection will be the only type of DI available to you.

The `BeanFactory` supports both of these variants for injecting dependencies into beans it manages. (It in fact also supports injecting setter-based dependencies after some dependencies have already been supplied via the constructor approach.) The configuration for the dependencies comes in the form of a `BeanDefinition`, which is used together with `PropertyEditor` instances to know how to convert properties from one format to another. However, most users of Spring will not be dealing with these classes directly (that is programmatically), but rather with an XML definition file which will be converted internally into instances of these classes, and used to load an entire Spring IoC container instance.

Bean dependency resolution generally happens as follows:

1. The `BeanFactory` is created and initialized with a configuration which describes all the beans. (Most Spring users use a `BeanFactory` or `ApplicationContext` implementation that supports XML format configuration files.)

2. Each bean has dependencies expressed in the form of properties, constructor arguments, or arguments to the static-factory method when that is used instead of a normal constructor. These dependencies will be provided to the bean, *when the bean is actually created*.

3. Each property or constructor argument is either an actual definition of the value to set, or a reference to another bean in the container.

4. Each property or constructor argument which is a value must be able to be converted from whatever format it was specified in, to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as `int`, `long`, `String`, `boolean`, etc.

The Spring container validates the configuration of each bean as the container is created, including the validation that properties which are bean references are actually referring to valid beans. However, the bean properties themselves are not set until the bean *is actually created*. For those beans that are singleton-scoped and set to be pre-instantiated (such as singleton beans in an `ApplicationContext`), creation happens at the time that the container is created, but otherwise this is only when the bean is requested. When a bean actually has to be created, this will potentially cause a graph of other beans to be created, as its dependencies and its dependencies' dependencies (and so on) are created and assigned.

Circular dependencies

If you are using predominantly constructor injection it is possible to write and configure your classes and beans such that an unresolvable circular dependency scenario is created.

Consider the scenario where you have class A, which requires an instance of class B to be provided via constructor injection, and class B, which requires an instance of class A to be provided via constructor injection. If you configure beans for classes A and B to be injected into each other, the Spring IoC container will detect this circular reference at runtime, and throw a `BeanCurrentlyInCreationException`.

One possible solution to this issue is to edit the source code of some of your classes to be configured via setters instead of via constructors. Another solution is not to use constructor injection and stick to setter injection only.

You can generally trust Spring to do the right thing. It will detect mis-configuration issues, such as references to non-existent beans and circular dependencies, at container load-time. It will actually set properties and resolve dependencies as late as possible, which is when the bean is actually created. This means that a Spring container which has loaded correctly can later generate an exception when you request a bean if there is a

problem creating that bean or one of its dependencies. This could happen if the bean throws an exception as a result of a missing or invalid property, for example. This potentially delayed visibility of some configuration issues is why `ApplicationContext` implementations by default pre-instantiate singleton beans. At the cost of some upfront time and memory to create these beans before they are actually needed, you find out about configuration issues when the `ApplicationContext` is created, not later. If you wish, you can still override this default behavior and set any of these singleton beans to lazy-initialize (that is not be pre-instantiated).

Finally, if it is not immediately apparent, it is worth mentioning that when one or more collaborating beans are being injected into a dependent bean, each collaborating bean is *totally* configured prior to being passed (via one of the DI flavors) to the dependent bean. This means that if bean A has a dependency on bean B, the Spring IoC container will *totally* configure bean B prior to invoking the setter method on bean A; you can read '*totally configure*' to mean that the bean will be instantiated (if not a pre-instantiated singleton), all of its dependencies will be set, and the relevant lifecycle methods (such as a configured init method or the IntializingBean callback method) will all be invoked.

3.3.1.3. Some examples

First, an example of using XML-based configuration metadata for setter-based DI. Find below a small part of a Spring XML configuration file specifying some bean definitions.

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- setter injection using the nested <ref/> element -->
  <property name="beanOne"><ref bean="anotherExampleBean"/></property>

  <!-- setter injection using the neater 'ref' attribute -->
  <property name="beanTwo" ref="yetAnotherBean"/>
  <property name="integerProperty" value="1"/>
</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.

Now, an example of using constructor-based DI. Find below a snippet from an XML configuration that specifies constructor arguments, and the corresponding Java class.

```
<bean id="exampleBean" class="examples.ExampleBean">

  <!-- constructor injection using the nested <ref/> element -->
  <constructor-arg><ref bean="anotherExampleBean"/></constructor-arg>

  <!-- constructor injection using the neater 'ref' attribute -->
  <constructor-arg ref="yetAnotherBean"/>
```

```
  <constructor-arg type="int" value="1"/>
</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`.

Now consider a variant of this where instead of using a constructor, Spring is told to call a `static` factory method to return an instance of the object:

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

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

```
public class ExampleBean {

    // a private constructor
    private ExampleBean(...) {
      ...
    }

    // a static factory method; the arguments to this method can be
    // considered the dependencies of the bean that is returned,
    // regardless of how those arguments are actually used.
    public static ExampleBean createInstance (
            AnotherBean anotherBean, YetAnotherBean yetAnotherBean, int i) {

        ExampleBean eb = new ExampleBean (...);
        // some other operations...
        return eb;
    }
}
```

Note that arguments to the `static` factory method are supplied via `constructor-arg` elements, exactly the same as if a constructor had actually been used. Also, it is important to realize that the type of the class being returned by the factory method does not have to be of the same type as the class which contains the `static` factory method, although in this example it is. An instance (non-static) factory method would be used in an essentially identical fashion (aside from the use of the `factory-bean` attribute instead of the `class` attribute), so details will not be discussed here.

3.3.2. Constructor Argument Resolution

Constructor argument resolution matching occurs using the argument's type. If there is no potential for ambiguity in the constructor arguments of a bean definition, then the order in which the constructor arguments are defined in a bean definition is the order in which those arguments will be supplied to the appropriate constructor when it is being instantiated. Consider the following class:

```
package x.y;

public class Foo {

    public Foo(Bar bar, Baz baz) {
        // ...
    }
}
```

There is no potential for ambiguity here (assuming of course that `Bar` and `Baz` classes are not related in an inheritance hierarchy). Thus the following configuration will work just fine, and you do not need to specify the constructor argument indexes and / or types explicitly.

```
<beans>
    <bean name="foo" class="x.y.Foo">
        <constructor-arg>
            <bean class="x.y.Bar"/>
        </constructor-arg>
        <constructor-arg>
            <bean class="x.y.Baz"/>
        </constructor-arg>
    </bean>
</beans>
```

When another bean is referenced, the type is known, and matching can occur (as was the case with the preceding example). When a simple type is used, such as `<value>true<value>`, Spring cannot determine the type of the value, and so cannot match by type without help. Consider the following class:

```
package examples;

public class ExampleBean {

    // No. of years to the calculate the Ultimate Answer
    private int years;

    // The Answer to Life, the Universe, and Everything
    private String ultimateAnswer;

    public ExampleBean(int years, String ultimateAnswer) {
        this.years = years;
        this.ultimateAnswer = ultimateAnswer;
    }
}
```

3.3.2.1. Constructor Argument Type Matching

The above scenario *can* use type matching with simple types by explicitly specifying the type of the constructor argument using the `'type'` attribute. For example:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg type="int" value="7500000"/>
  <constructor-arg type="java.lang.String" value="42"/>
</bean>
```

3.3.2.2. Constructor Argument Index

Constructor arguments can have their index specified explicitly by use of the `index` attribute. For example:

```
<bean id="exampleBean" class="examples.ExampleBean">
  <constructor-arg index="0" value="7500000"/>
  <constructor-arg index="1" value="42"/>
</bean>
```

As well as solving the ambiguity problem of multiple simple values, specifying an index also solves the problem of ambiguity where a constructor may have two arguments of the same type. Note that the *index is 0 based*.

3.3.3. Bean properties and constructor arguments detailed

As mentioned in the previous section, bean properties and constructor arguments can be defined as either references to other managed beans (collaborators), or values defined inline. Spring's XML-based configuration metadata supports a number of sub-element types within its `<property/>` and `<constructor-arg/>` elements for just this purpose.

3.3.3.1. Straight values (primitives, `Strings`, etc.)

The `<value/>` element specifies a property or constructor argument as a human-readable string representation. As mentioned previously, JavaBeans `PropertyEditors` are used to convert these string values from a `String` to the actual type of the property or argument.

```
<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>
  <property name="password">
    <value>masterkaoli</value>
  </property>
</bean>
```

The `<property/>` and `<constructor-arg/>` elements also support the use of the `'value'` attribute, which can lead to much more succinct configuration. When using the `'value'` attribute, the above bean definition reads like so:

```
<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"/>
  <property name="url" value="jdbc:mysql://localhost:3306/mydb"/>
  <property name="username" value="root"/>
  <property name="password" value="masterkaoli"/>
</bean>
```

The Spring team generally prefer the attribute style over the use of nested `<value/>` elements. If you are reading this reference manual straight through from top to bottom (wow!) then we are getting slightly ahead of ourselves here, but you can also configure a `java.util.Properties` instance like so:

```
<bean id="mappings" class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">

   <!-- typed as a java.util.Properties -->
   <property name="properties">
      <value>
```

```
        jdbc.driver.className=com.mysql.jdbc.Driver
        jdbc.url=jdbc:mysql://localhost:3306/mydb
      </value>
   </property>
</bean>
```

Can you see what is happening? The Spring container is converting the text inside the `<value/>` element into a `java.util.Properties` instance using the JavaBeans `PropertEditor` mechanism. This is a nice shortcut, and is one of a few places where the Spring team do favor the use of the nested `<value/>` element over the `'value'` attribute style.

3.3.3.1.1. The `idref` element

The `idref` element is simply an error-proof way to pass the *id* of another bean in the container (to a `<constructor-arg/>` or `<property/>` element).

```
<bean id="theTargetBean" class="..."/>

<bean id="theClientBean" class="...">
    <property name="targetName">
        <idref bean="theTargetBean" />
    </property>
</bean>
```

The above bean definition snippet is *exactly* equivalent (at runtime) to the following snippet:

```
<bean id="theTargetBean" class="..."/>

<bean id="client" class="...">
    <property name="targetName">
        <value>theTargetBean</value>
    </property>
</bean>
```

The main reason the first form is preferable to the second is that using the `idref` tag allows the container to validate *at deployment time* that the referenced, named bean actually exists. In the second variation, no validation is performed on the value that is passed to the `'targetName'` property of the `'client'` bean. Any typo will only be discovered (with most likely fatal results) when the `'client'` bean is actually instantiated. If the `'client'` bean is a prototype bean, this typo (and the resulting exception) may only be discovered long after the container is actually deployed.

Additionally, if the bean being referred to is in the same XML unit, and the bean name is the bean *id*, the `'local'` attribute may be used, which allows the XML parser itself to validate the bean id even earlier, at XML document parse time.

```
<property name="targetName">
   <!-- a bean with an id of 'theTargetBean' must exist,
                otherwise an XML exception will be thrown -->
   <idref local="theTargetBean"/>
</property>
```

By way of an example, one common place (at least in pre-Spring 2.0 configuration) where the <idref/> element brings value is in the configuration of AOP interceptors in a `ProxyFactoryBean` bean definition. If you use <idref/> elements when specifying the interceptor names, there is no chance of inadvertently misspelling an interceptor id.

3.3.3.2. References to other beans (collaborators)

The `ref` element is the final element allowed inside a `<constructor-arg/>` or `<property/>` definition element.

It is used to set the value of the specified property to be a reference to another bean managed by the container (a collaborator). As mentioned in a previous section, the referred-to bean is considered to be a dependency of the bean who's property is being set, and will be initialized on demand as needed (if it is a singleton bean it may have already been initialized by the container) before the property is set. All references are ultimately just a reference to another object, but there are 3 variations on how the id/name of the other object may be specified, which determines how scoping and validation is handled.

Specifying the target bean by using the `bean` attribute of the `<ref/>` tag is the most general form, and will allow creating a reference to any bean in the same container (whether or not in the same XML file), or parent container. The value of the `'bean'` attribute may be the same as either the `'id'` attribute of the target bean, or one of the values in the `'name'` attribute of the target bean.

```
<ref bean="someBean"/>
```

Specifying the target bean by using the `local` attribute leverages the ability of the XML parser to validate XML id references within the same file. The value of the `local` attribute must be the same as the `id` attribute of the target bean. The XML parser will issue an error if no matching element is found in the same file. As such, using the local variant is the best choice (in order to know about errors as early as possible) if the target bean is in the same XML file.

```
<ref local="someBean"/>
```

Specifying the target bean by using the `'parent'` attribute allows a reference to be created to a bean which is in a parent container of the current container. The value of the `'parent'` attribute may be the same as either the `'id'` attribute of the target bean, or one of the values in the `'name'` attribute of the target bean, and the target bean must be in a parent container to the current one. The main use of this bean reference variant is when you have a hierarchy of containers and you want to wrap an existing bean in a parent container with some sort of proxy which will have the same name as the parent bean.

```
<!-- in the parent context -->
<bean id="accountService" class="com.foo.SimpleAccountService">
    <!-- insert dependencies as required as here -->
</bean>
```

```
<!-- in the child (descendant) context -->
<bean id="accountService"  <-- notice that the name of this bean is the same as the name of the 'parent' bean
      class="org.springframework.aop.framework.ProxyFactoryBean">
      <property name="target">
          <ref parent="accountService"/>  <-- notice how we refer to the parent bean
      </property>
    <!-- insert other configuration and dependencies as required as here -->
</bean>
```

3.3.3.3. Inner beans

A `<bean/>` element inside the `<property/>` or `<constructor-arg/>` elements is used to define a so-called *inner bean*. An inner bean definition does not need to have any id or name defined, and it is best not to even specify any id or name value because the id or name value simply will be ignored by the container.

```
<bean id="outer" class="...">
  <!-- instead of using a reference to a target bean, simply define the target bean inline -->
  <property name="target">
    <bean class="com.mycompany.Person"> <!-- this is the inner bean -->
      <property name="name" value="Fiona Apple"/>
      <property name="age" value="25"/>
    </bean>
  </property>
</bean>
```

Note that in the specific case of inner beans, the `'scope'` flag and any `'id'` or `'name'` attribute are effectively ignored. Inner beans are *always* anonymous and they are *always* scoped as prototypes. Please also note that it is *not* possible to inject inner beans into collaborating beans other than the enclosing bean.

3.3.3.4. Collections

The `<list/>`, `<set/>`, `<map/>`, and `<props/>` elements allow properties and arguments of the Java `Collection` type `List`, `Set`, `Map`, and `Properties`, respectively, to be defined and set.

```
<bean id="moreComplexObject" class="example.ComplexObject">
  <!-- results in a setAdminEmails(java.util.Properties) call -->
  <property name="adminEmails">
    <props>
        <prop key="administrator">administrator@somecompany.org</prop>
        <prop key="support">support@somecompany.org</prop>
        <prop key="development">development@somecompany.org</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>
                <value>yup an entry</value>
            </key>
            <value>just some string</value>
        </entry>
        <entry>
            <key>
                <value>yup a ref</value>
            </key>
            <ref bean="myDataSource" />
        </entry>
    </map>
  </property>
  <!-- results in a setSomeSet(java.util.Set) call -->
  <property name="someSet">
    <set>
        <value>just some string</value>
        <ref bean="myDataSource" />
    </set>
  </property>
</bean>
```

Note that the value of a map key or value, or a set value, can also again be any of the following elements:

```
bean | ref | idref | list | set | map | props | value | null
```

3.3.3.4.1. Collection merging

As of Spring 2.0, the container also supports the *merging* of collections. This allows an application developer to define a parent-style `<list/>`, `<map/>`, `<set/>` or `<props/>` element, and have child-style `<list/>`, `<map/>`, `<set/>` or `<props/>` elements inherit and override values from the parent collection; that is to say the child collection's values will be the result obtained from the merging of the elements of the parent and child collections, with the child's collection elements overriding values specified in the parent collection.

Please note that this section on merging makes use of the parent-child bean mechanism. This concept has not yet been introduced, so readers unfamiliar with the concept of parent and child bean definitions may wish to

read the relevant section before continuing.

Find below an example of the collection merging feature:

```
<beans>
<bean id="parent" abstract="true" class="example.ComplexObject">
    <property name="adminEmails">
        <props>
            <prop key="administrator">administrator@somecompany.com</prop>
            <prop key="support">support@somecompany.com</prop>
        </props>
    </property>
</bean>
<bean id="child" parent="parent">
    <property name="adminEmails">
        <!-- the merge is specified on the *child* collection definition -->
        <props merge="true">
            <prop key="sales">sales@somecompany.com</prop>
            <prop key="support">support@somecompany.co.uk</prop>
        </props>
    </property>
</bean>
<beans>
```

Notice the use of the `merge=true` attribute on the `<props/>` element of the `adminEmails` property of the `child` bean definition. When the `child` bean is actually resolved and instantiated by the container, the resulting instance will have an `adminEmails` `Properties` collection that contains the result of the merging of the child's `adminEmails` collection with the parent's `adminEmails` collection.

```
administrator=administrator@somecompany.com
sales=sales@somecompany.com
support=support@somecompany.co.uk
```

Notice how the child `Properties` collection's value set will have inherited all the property elements from the parent `<props/>`. Notice also how the child's value for the `support` value overrides the value in the parent collection.

This merging behavior applies similarly to the `<list/>`, `<map/>`, and `<set/>` collection types. In the specific case of the `<list/>` element, the semantics associated with the `List` collection type, that is the notion of an `ordered` collection of values, is maintained; the parent's values will precede all of the child list's values. In the case of the `Map`, `Set`, and `Properties` collection types, there is no notion of ordering and hence no ordering semantics are in effect for the collection types that underlie the associated `Map`, `Set` and `Properties` implementation types used internally by the container.

Finally, some minor notes about the merging support are in order; you cannot merge different collection types (e.g. a `Map` and a `List`), and if you do attempt to do so an appropriate `Exception` will be thrown; and in case it is not immediately obvious, the `'merge'` attribute must be specified on the lower level, inherited, child definition; specifying the `'merge'` attribute on a parent collection definition is redundant and will not result in the desired merging; and (lastly), please note that this merging feature is only available in Spring 2.0 (and later versions).

3.3.3.4.2. Strongly-typed collection (Java 5+ only)

If you are using Java 5 or Java 6, you will be aware that it is possible to have strongly typed collections (using generic types). That is, it is possible to declare a `Collection` type such that it can only contain `String` elements (for example). If you are using Spring to dependency inject a strongly-typed `Collection` into a bean, you can take advantage of Spring's type-conversion support such that the elements of your strongly-typed `Collection` instances will be converted to the appropriate type prior to being added to the `Collection`.

```
public class Foo {
```

```
    private Map<String, Float> accounts;

    public void setAccounts(Map<String, Float> accounts) {
        this.accounts = accounts;
    }
}
```

```
<beans>
    <bean id="foo" class="x.y.Foo">
        <property name="accounts">
            <map>
                <entry key="one" value="9.99"/>
                <entry key="two" value="2.75"/>
                <entry key="six" value="3.99"/>
            </map>
        </property>
    </bean>
</beans>
```

When the `'accounts'` property of the `'foo'` bean is being prepared for injection, the generics information about the element type of the strongly-typed `Map<String, Float>` is actually available via reflection, and so Spring's type conversion infrastructure will actually recognize the various value elements as being of type `Float` and so the string values `'9.99'`, `'2.75'`, and `'3.99'` will be converted into an actual `Float` type.

3.3.3.5. `Nulls`

The `<null/>` element is used to handle `null` values. Spring treats empty arguments for properties and the like as empty `Strings`. The following XML-based configuration metadata snippet results in the email property being set to the empty `String` value ("")

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

This is equivalent to the following Java code: `exampleBean.setEmail("")`. The special `<null>` element may be used to indicate a `null` value. For example:

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

The above configuration is equivalent to the following Java code: `exampleBean.setEmail(null)`.

3.3.3.6. Shortcuts and other convenience options for XML-based configuration metadata

The configuration metadata shown so far is a tad verbose. That is why there are several options available for you to limit the amount of XML you have to write to configure your components. The first is a shortcut to define values and references to other beans as part of a `<property/>` definition. The second is slightly different format of specifying properties alltogether.

3.3.3.6.1. XML-based configuration metadata shortcuts

The `<property/>`, `<constructor-arg/>`, and `<entry/>` elements all support a `'value'` attribute which may be used instead of embedding a full `<value/>` element. Therefore, the following:

```
<property name="myProperty">
  <value>hello</value>
</property>
```

```
<constructor-arg>
  <value>hello</value>
</constructor-arg>
```

```
<entry key="myKey">
  <value>hello</value>
</entry>
```

are equivalent to:

```
<property name="myProperty" value="hello"/>
```

```
<constructor-arg value="hello"/>
```

```
<entry key="myKey" value="hello"/>
```

The `<property/>` and `<constructor-arg/>` elements support a similar shortcut `'ref'` attribute which may be used instead of a full nested `<ref/>` element. Therefore, the following:

```
<property name="myProperty">
  <ref bean="myBean">
</property>
```

```
<constructor-arg>
  <ref bean="myBean">
</constructor-arg>
```

... are equivalent to:

```
<property name="myProperty" ref="myBean"/>
```

```
<constructor-arg ref="myBean"/>
```

Note however that the shortcut form is equivalent to a `<ref bean="xxx">` element; there is no shortcut for `<ref local="xxx">`. To enforce a strict local reference, you must use the long form.

Finally, the entry element allows a shortcut form to specify the key and/or value of the map, in the form of the `'key'` / `'key-ref'` and `'value'` / `'value-ref'` attributes. Therefore, the following:

```
<entry>
  <key>
    <ref bean="myKeyBean" />
  </key>
  <ref bean="myValueBean" />
</entry>
```

is equivalent to:

```
<entry key-ref="myKeyBean" value-ref="myValueBean"/>
```

Again, the shortcut form is equivalent to a `<ref bean="xxx">` element; there is no shortcut for `<ref local="xxx">`.

3.3.3.6.2. The p-namespace and how to use it to configure properties

The second option you have to limit the amount of XML you have to write to configure your components is to use the special "p-namespace". Spring 2.0 and later features support for extensible configuration formats using namespaces. Those namespaces are all based on an XML Schema definition. In fact, the `beans` configuration format that you've been reading about is defined in an XML Schema document.

One special namespace is not defined in an XSD file, and only exists in the core of Spring itself. The so-called p-namespace doesn't need a schema definition and is an alternative way of configuring your properties differently than the way you have seen so far. Instead of using **nested** `property` elements, using the p-namespace you can use attributes as part of the `bean` element that describe your property values. The values of the attributes will be taken as the values for your properties.

The following two XML snippets boil down to the same thing in the end: the first is using the format you're familiar with (the `property` elements) whereas the second example is using the p-namespace

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean name="classic" class="com.mycompany.ExampleBean">
        <property name="email" value="foo@bar.com/>
    </bean>

    <bean name="p-namespace"
        class="com.mycompany.ExampleBean"
        p:email="foo@bar.com"/>
</beans>
```

As you can see, we are including an attribute from the p-namespace called email in the bean definition. This is telling Spring that it should include a property declaration. As previously mentioned, the p-namespace doesn't have a schema definition, so the name of the attribute can be set to whatever name your property has.

This next example includes two more bean definitions that both have a reference to another bean:

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean name="john-classic" class="com.mycompany.Person">
        <property name="name" value="John Doe"/>
        <property name="spouse" ref="jane"/>
    </bean>

    <bean name="john-modern"
        class="com.mycompany.Person"
        p:name="John Doe"
        p:spouse-ref="jane"/>

    <bean name="jane" class="com.mycompany.Person">
        <property name="name" value="Jane Doe"/>
    </bean>
</beans>
```

As you can see, this example doesn't only include a property value using the p-namespace, but also uses a special format to declare property references. Whereas the first bean definition uses `<property name="spouse" ref="jane"/>` to create a reference from bean john to bean jane, the second bean definition uses `p:spouse-ref="jane"` as an attribute to do the exact same thing. In this case **spouse** is the property name

whereas the `-ref` part tells Spring this is not a value but a bean reference.

Note

Note that we recommend you to choose carefully which approach you are going to use in your project. You should also communicate this to your team members so you won't end up with XML documents using all three approaches at the same time. This will prevent people from not understanding the application because of different ways of configuring it, and will add to the consistency of your codebase. Also note that this functionality is only available as of Spring 2.0.

3.3.3.7. Compound property names

Compound or nested property names are perfectly legal when setting bean properties, as long as all components of the path except the final property name are not null. For example, in this bean definition:

```
<bean id="foo" class="foo.Bar">
  <property name="fred.bob.sammy" value="123" />
</bean>
```

The `foo` bean has a `fred` property which has a `bob` property, which has a `sammy` property, and that final `sammy` property is being set to the value `123`. In order for this to work, the `fred` property of `foo`, and the `bob` property of `fred` must both be non-null after the bean is constructed, or a `NullPointerException` will be thrown.

3.3.4. Using `depends-on`

For most situations, the fact that a bean is a dependency of another is expressed by the fact that one bean is set as a property of another. This is typically accomplished with the `<ref/>` element in XML-based configuration metadata. For the relatively infrequent situations where dependencies between beans are less direct (for example, when a static initializer in a class needs to be triggered, such as database driver registration), the `'depends-on'` attribute may be used to explicitly force one or more beans to be initialized before the bean using this element is initialized. Find below an example of using the `'depends-on'` attribute to express a dependency on a single bean.

```
<bean id="beanOne" class="ExampleBean" depends-on="manager"/>

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

If you need to express a dependency on multiple beans, you can supply a list of bean names as the value of the `'depends-on'` attribute, with commas, whitespace and semi-colons all valid delimiters, like so:

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

<bean id="manager" class="ManagerBean" />
<bean id="accountDao" class="x.y.jdbc.JdbcAccountDao" />
```

Note

The `'depends-on'` attribute and property is used not only to specify an initialization time dependency, but also to specify the corresponding destroy time dependency (in the case of singleton beans only). Dependant beans that are defined in the `'depends-on'` attribute will be destroyed first prior to the relevant bean itself being destroyed. This thus allows you to control shutdown order too.

3.3.5. Lazily-instantiated beans

The default behavior for `ApplicationContext` implementations is to eagerly pre-instantiate all `singleton` beans at startup. Pre-instantiation means that an `ApplicationContext` will eagerly create and configure all of its singleton beans as part of its initialization process. Generally this is *a good thing*, because it means that any errors in the configuration or in the surrounding environment will be discovered immediately (as opposed to possibly hours or even days down the line).

However, there are times when this behavior is *not* what is wanted. If you do not want a singleton bean to be pre-instantiated when using an `ApplicationContext`, you can selectively control this by marking a bean definition as lazy-initialized. A lazily-initialized bean indicates to the IoC container whether or not a bean instance should be created at startup or when it is first requested.

When configuring beans via XML, this lazy loading is controlled by the `'lazy-init'` attribute on the `<bean/>` element; for example:

```
<bean id="lazy" class="com.foo.ExpensiveToCreateBean" lazy-init="true"/>

<bean name="not.lazy" class="com.foo.AnotherBean"/>
```

When the above configuration is consumed by an `ApplicationContext`, the bean named `'lazy'` will *not* be eagerly pre-instantiated when the `ApplicationContext` is starting up, whereas the `'not.lazy'` bean will be eagerly pre-instantiated.

One thing to understand about lazy-initialization is that even though a bean definition may be marked up as being lazy-initialized, if the lazy-initialized bean is the dependency of a singleton bean that is not lazy-initialized, when the `ApplicationContext` is eagerly pre-instantiating the singleton, it will have to satisfy all of the singletons dependencies, one of which will be the lazy-initialized bean! So don't be confused if the IoC container creates one of the beans that you have explicitly configured as lazy-initialized at startup; all that means is that the lazy-initialized bean is being injected into a non-lazy-initialized singleton bean elsewhere.

It is also possible to control lazy-initialization at the container level by using the `'default-lazy-init'` attribute on the `<beans/>` element; for example:

```
<beans default-lazy-init="true">
    <!-- no beans will be pre-instantiated... -->
</beans>
```

3.3.6. Autowiring collaborators

The Spring container is able to *autowire* relationships between collaborating beans. This means that it is possible to automatically let Spring resolve collaborators (other beans) for your bean by inspecting the contents of the `BeanFactory`. The autowiring functionality has five modes. Autowiring is specified *per* bean and can thus be enabled for some beans, while other beans will not be autowired. Using autowiring, it is possible to reduce or eliminate the need to specify properties or constructor arguments, thus saving a significant amount of typing. [2] When using XML-based configuration metadata, the autowire mode for a bean definition is specified by using the `autowire` attribute of the `<bean/>` element. The following values are allowed:

Table 3.2. Autowiring modes

Mode	Explanation
no	

[2] See the section entitled Section 3.3.1, “Injecting dependencies”

Mode	Explanation
	No autowiring at all. Bean references must be defined via a `ref` element. This is the default, and changing this is discouraged for larger deployments, since explicitly specifying collaborators gives greater control and clarity. To some extent, it is a form of documentation about the structure of a system.
byName	Autowiring by property name. This option will inspect the container and look for a bean named exactly the same as the property which needs to be autowired. For example, if you have a bean definition which is set to autowire by name, and it contains a *master* property (that is, it has a *setMaster(..)* method), Spring will look for a bean definition named `master`, and use it to set the property.
byType	Allows a property to be autowired if there is exactly one bean of the property type in the container. If there is more than one, a fatal exception is thrown, and this indicates that you may not use *byType* autowiring for that bean. If there are no matching beans, nothing happens; the property is not set. If this is not desirable, setting the `dependency-check="objects"` attribute value specifies that an error should be thrown in this case.
constructor	This is analogous to *byType*, but applies to constructor arguments. If there isn't exactly one bean of the constructor argument type in the container, a fatal error is raised.
autodetect	Chooses *constructor* or *byType* through introspection of the bean class. If a default constructor is found, the *byType* mode will be applied.

Note that explicit dependencies in `property` and `constructor-arg` settings `always` override autowiring. Please also note that it is not currently possible to autowire so-called *simple* properties such as primitives, `Strings`, and `Classes` (and arrays of such simple properties).(This is by-design and should be considered a *feature*.) Autowire behavior can be combined with dependency checking, which will be performed after all autowiring has been completed.

It is important to understand the various advantages and disadvantages of autowiring. Some advantages of autowiring include:

- Autowiring can significantly reduce the volume of configuration required. However, mechanisms such as the use of a bean template (discussed elsewhere in this chapter) are also valuable in this regard.
- Autowiring can cause configuration to keep itself up to date as your objects evolve. For example, if you need to add an additional dependency to a class, that dependency can be satisfied automatically without the need to modify configuration. Thus there may be a strong case for autowiring during development, without ruling out the option of switching to explicit wiring when the code base becomes more stable.

Some disadvantages of autowiring:

- Autowiring is more magical than explicit wiring. Although, as noted in the above table, Spring is careful to avoid guessing in case of ambiguity which might have unexpected results, the relationships between your Spring-managed objects are no longer documented explicitly.
- Wiring information may not be available to tools that may generate documentation from a Spring container.

- Autowiring by type will only work when there is a single bean definition of the type specified by the setter method or constructor argument. You need to use explicit wiring if there is any potential ambiguity.

There is no wrong or right answer in all cases. A degree of consistency across a project is best though; for example, if autowiring is not used in general, it might be confusing to developers to use it just to wire one or two bean definitions.

3.3.6.1. Excluding a bean from being available for autowiring

You can also (on a per-bean basis) totally exclude a bean from being an autowire candidate. When configuring beans using Spring's XML format, the `'autowire-candidate'` attribute of the `<bean/>` element can be set to `'false'`; this has the effect of making the container totally exclude that specific bean definition from being available to the autowiring infrastructure.

This can be useful when you have a bean that you absolutely never ever want to have injected into other beans via autowiring. It does not mean that the excluded bean cannot itself be configured using autowiring... it can, it is rather that it itself will not be considered as a candidate for autowiring other beans.

3.3.7. Checking for dependencies

The Spring IoC container also has the ability to check for the existence of unresolved dependencies of a bean deployed into the container. These are JavaBeans properties of the bean, which do not have actual values set for them in the bean definition, or alternately provided automatically by the autowiring feature.

This feature is sometimes useful when you want to ensure that all properties (or all properties of a certain type) are set on a bean. Of course, in many cases a bean class will have default values for many properties, or some properties do not apply to all usage scenarios, so this feature is of limited use. Dependency checking can also be enabled and disabled per bean, just as with the autowiring functionality. The default is to *not* check dependencies. Dependency checking can be handled in several different modes. When using XML-based configuration metadata, this is specified via the `'dependency-check'` attribute in a bean definition, which may have the following values.

Table 3.3. Dependency checking modes

Mode	Explanation
none	No dependency checking. Properties of the bean which have no value specified for them are simply not set.
simple	Dependency checking is performed for primitive types and collections (everything except collaborators).
object	Dependency checking is performed for collaborators only.
all	Dependency checking is done for collaborators, primitive types and collections.

If you are using Java 5 and thus have access to source-level annotations, you may find the section entitled Section 25.3.1, “`@Required`” to be of interest.

3.3.8. Method Injection

For most application scenarios, the majority of the beans in the container will be singletons. When a singleton bean needs to collaborate with another singleton bean, or a non-singleton bean needs to collaborate with another non-singleton bean, the typical and common approach of handling this dependency by defining one bean to be a property of the other is quite adequate. There is a problem when the bean lifecycles are different. Consider a singleton bean A which needs to use a non-singleton (prototype) bean B, perhaps on each method invocation on A. The container will only create the singleton bean A once, and thus only get the opportunity to set the properties once. There is no opportunity for the container to provide bean A with a new instance of bean B every time one is needed.

One solution to this issue is to forgo some inversion of control. Bean A can be made aware of the container by implementing the `BeanFactoryAware` interface, and use programmatic means to ask the container via a `getBean("B")` call for (a typically new) bean B instance every time it needs it. Find below an admittedly somewhat contrived example of this approach:

```
// a class that uses a stateful Command-style class to perform some processing
package fiona.apple;

// lots of Spring-API imports
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.beans.factory.BeanFactoryAware;

public class CommandManager implements BeanFactoryAware {

   private BeanFactory beanFactory;

   public Object process(Map commandState) {
      // grab a new instance of the appropriate Command
      Command command = createCommand();
      // set the state on the (hopefully brand new) Command instance
      command.setState(commandState);
      return command.execute();
   }

   // the Command returned here could be an implementation that executes asynchronously, or whatever
   protected Command createCommand() {
      return (Command) this.beanFactory.getBean("command"); // notice the Spring API dependency
   }

   public void setBeanFactory(BeanFactory beanFactory) throws BeansException {
      this.beanFactory = beanFactory;
   }
}
```

The above example is generally not a desirable solution since the business code is then aware of and coupled to the Spring Framework. Method Injection, a somewhat advanced feature of the Spring IoC container, allows this use case to be handled in a clean fashion.

3.3.8.1. Lookup method injection

> **Isn't this Method Injection...**
>
> ... somewhat like Tapestry 4.0's pages, where folks wrote `abstract` properties that Tapestry would override at runtime with implementations that did stuff? It sure is (well, kinda).
>
> You can read more about the motivation for Method Injection in this blog entry.

Lookup method injection refers to the ability of the container to override methods on *container managed beans*,

to return the result of looking up another named bean in the container. The lookup will typically be of a prototype bean as in the scenario described above. The Spring Framework implements this method injection by dynamically generating a subclass overriding the method, using bytecode generation via the CGLIB library.

So if you look at the code from previous code snippet (the `CommandManager` class), the Spring container is going to dynamically override the implementation of the `createCommand()` method. Your `CommandManager` class is not going to have any Spring dependencies, as can be seen in this reworked example below:

```
package fiona.apple;

// no more Spring imports!

public abstract class CommandManager {

   public Object process(Object commandState) {
      // grab a new instance of the appropriate Command interface
      Command command = createCommand();
      // set the state on the (hopefully brand new) Command instance
      command.setState(commandState);
      return command.execute();
   }

    // okay... but where is the implementation of this method?
   protected abstract Command createCommand();
}
```

In the client class containing the method to be injected (the `CommandManager` in this case), the method that is to be 'injected' must have a signature of the following form:

```
<public|protected> [abstract] <return-type> theMethodName(no-arguments);
```

If the method is `abstract`, the dynamically-generated subclass will implement the method. Otherwise, the dynamically-generated subclass will override the concrete method defined in the original class. Let's look at an example:

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="command" class="fiona.apple.AsyncCommand" scope="prototype">
  <!-- inject dependencies here as required -->
</bean>

<!-- commandProcessor uses statefulCommandHelper -->
<bean id="commandManager" class="fiona.apple.CommandManager">
  <lookup-method name="createCommand" bean="command"/>
</bean>
```

The bean identified as *commandManager* will call its own method `createCommand()` whenever it needs a new instance of the *command* bean. It is important to note that the person deploying the beans must be careful to deploy the `command` bean as a prototype (if that is actually what is needed). If it is deployed as a singleton, the same instance of the `command` bean will be returned each time!

Please be aware that in order for this dynamic subclassing to work, you will need to have the CGLIB jar(s) on your classpath. Additionally, the class that the Spring container is going to subclass cannot be `final`, and the method that is being overridden cannot be `final` either. Also, testing a class that has an `abstract` method can be somewhat odd in that you will have to subclass the class yourself and supply a stub implementation of the `abstract` method. Finally, objects that have been the target of method injection cannot be serialized.

Tip

The interested reader may also find the `ServiceLocatorFactoryBean` (in the `org.springframework.beans.factory.config` package) to be of use; the approach is similar to that of the `ObjectFactoryCreatingFactoryBean`, but it allows you to specify your own lookup

interface as opposed to having to use a Spring-specific lookup interface such as the `ObjectFactory`. Consult the (copious) Javadocs for the `ServiceLocatorFactoryBean` for a full treatment of this alternative approach (that *does* reduce the coupling to Spring).

3.3.8.2. Arbitrary method replacement

A less commonly useful form of method injection than Lookup Method Injection is the ability to replace arbitrary methods in a managed bean with another method implementation. Users may safely skip the rest of this section (which describes this somewhat advanced feature), until this functionality is actually needed.

When using XML-based configuration metadata, the `replaced-method` element may be used to replace an existing method implementation with another, for a deployed bean. Consider the following class, with a method computeValue, which we want to override:

```
public class MyValueCalculator {

  public String computeValue(String input) {
    // some real code...
  }

  // some other methods...

}
```

A class implementing the `org.springframework.beans.factory.support.MethodReplacer` interface provides the new method definition.

```
/** meant to be used to override the existing computeValue(String)
    implementation in MyValueCalculator
  */
public class ReplacementComputeValue implements MethodReplacer {

    public Object reimplement(Object o, Method m, Object[] args) throws Throwable {
        // get the input value, work with it, and return a computed result
        String input = (String) args[0];
        ...
        return ...;
    }
}
```

The bean definition to deploy the original class and specify the method override would look like this:

```
<bean id="myValueCalculator class="x.y.z.MyValueCalculator">
  <!-- arbitrary method replacement -->
  <replaced-method name="computeValue" replacer="replacementComputeValue">
    <arg-type>String</arg-type>
  </replaced-method>
</bean>

<bean id="replacementComputeValue" class="a.b.c.ReplacementComputeValue"/>
```

One or more contained `<arg-type/>` elements within the `<replaced-method/>` element may be used to indicate the method signature of the method being overridden. Note that the signature for the arguments is actually only needed in the case that the method is actually overloaded and there are multiple variants within the class. For convenience, the type string for an argument may be a substring of the fully qualified type name. For example, all the following would match `java.lang.String`.

```
    java.lang.String
    String
    Str
```

Since the number of arguments is often enough to distinguish between each possible choice, this shortcut can save a lot of typing, by allowing you to type just the shortest string that will match an argument type.

3.4. Bean scopes

When you create a bean definition what you are actually creating is a *recipe* for creating actual instances of the class defined by that bean definition. The idea that a bean definition is a recipe is important, because it means that, just like a class, you can potentially have many object instances created from a single recipe.

You can control not only the various dependencies and configuration values that are to be plugged into an object that is created from a particular bean definition, but also the *scope* of the objects created from a particular bean definition. This approach is very powerful and gives you the flexibility to *choose* the scope of the objects you create through configuration instead of having to 'bake in' the scope of an object at the Java class level. Beans can be defined to be deployed in one of a number of scopes: out of the box, the Spring Framework supports exactly five scopes (of which three are available only if you are using a web-aware `ApplicationContext`).

The scopes supported out of the box are listed below:

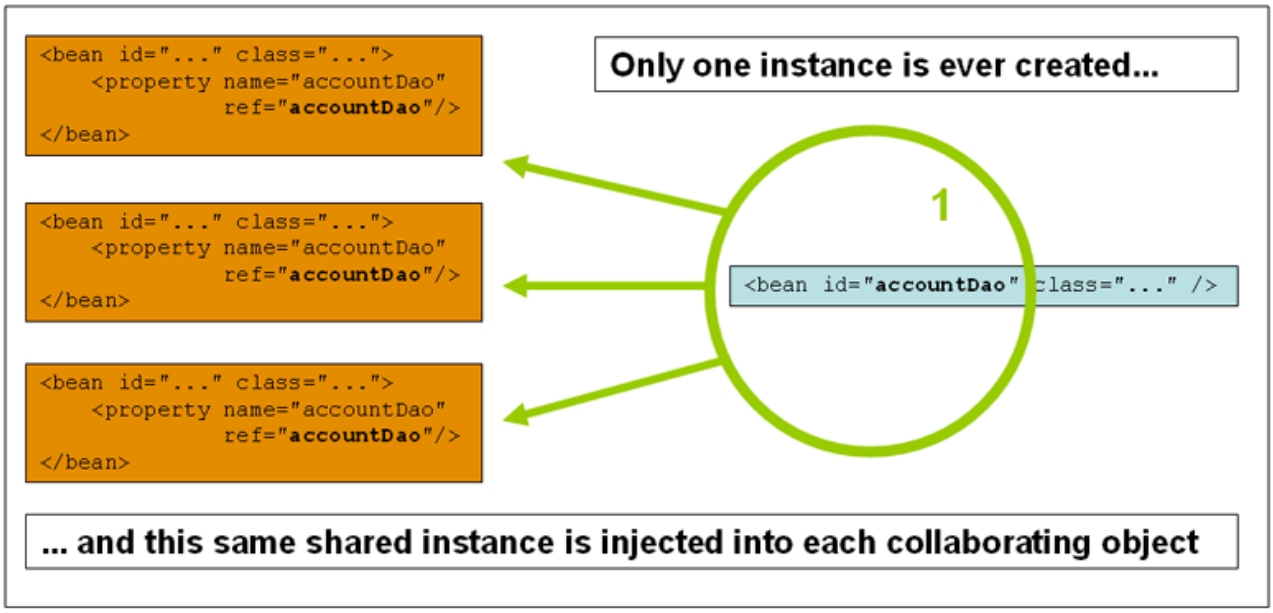
Table 3.4. Bean scopes

Scope	Description
singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring `ApplicationContext`.
session	Scopes a single bean definition to the lifecycle of a HTTP `Session`. Only valid in the context of a web-aware Spring `ApplicationContext`.
global session	Scopes a single bean definition to the lifecycle of a global HTTP `Session`. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring `ApplicationContext`.

3.4.1. The singleton scope

When a bean is a singleton, only one *shared* instance of the bean will be managed, and all requests for beans with an id or ids matching that bean definition will result in that one specific bean instance being returned by the Spring container.

To put it another way, when you define a bean definition and it is scoped as a singleton, then the Spring IoC container will create *exactly one* instance of the object defined by that bean definition. This single instance will be stored in a cache of such singleton beans, and *all subsequent requests and references* for that named bean will result in the cached object being returned.



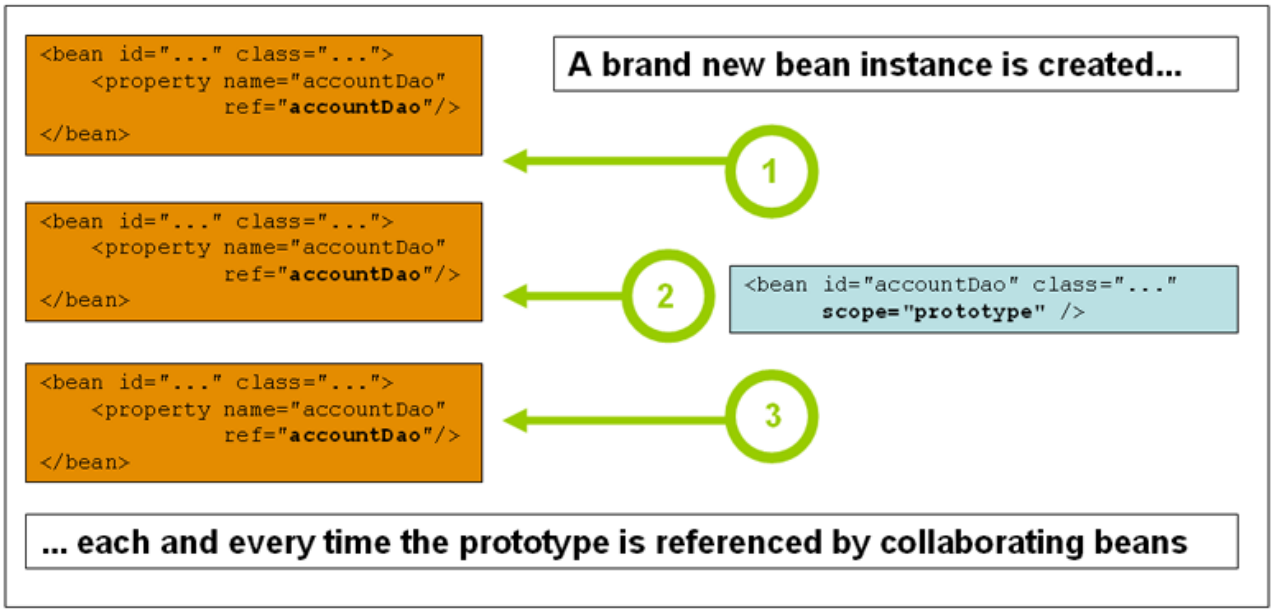
Please be aware that Spring's concept of a singleton bean is quite different from the Singleton pattern as defined in the seminal Gang of Four (GoF) patterns book. The GoF Singleton hardcodes the scope of an object such that one *and only one* instance of a particular class will ever be created *per* `ClassLoader`. The scope of the Spring singleton is best described as *per container and per bean*. This means that if you define one bean for a particular class in a single Spring container, then the Spring container will create one *and only one* instance of the class defined by that bean definition. *The singleton scope is the default scope in Spring*. To define a bean as a singleton in XML, you would write configuration like so:

```
<bean id="accountService" class="com.foo.DefaultAccountService"/>

<!-- the following is equivalent, though redundant (singleton scope is the default); using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="singleton"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="true"/>
```

3.4.2. The prototype scope

The non-singleton, prototype scope of bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is made (that is, it is injected into another bean or it is requested via a programmatic `getBean()` method call on the container). As a rule of thumb, you should use the prototype scope for all beans that are stateful, while the singleton scope should be used for stateless beans.

The following diagram illustrates the Spring prototype scope. *Please note that a DAO would not typically be configured as a prototype, since a typical DAO would not hold any conversational state; it was just easier for this author to reuse the core of the singleton diagram.*



To define a bean as a prototype in XML, you would write configuration like so:

```
<!-- using spring-beans-2.0.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" scope="prototype"/>

<!-- the following is equivalent and preserved for backward compatibility in spring-beans.dtd -->
<bean id="accountService" class="com.foo.DefaultAccountService" singleton="false"/>
```

There is one quite important thing to be aware of when deploying a bean in the prototype scope, in that the lifecycle of the bean changes slightly. Spring does not manage the complete lifecycle of a prototype bean: the container instantiates, configures, decorates and otherwise assembles a prototype object, hands it to the client and then has no further knowledge of that prototype instance. This means that while *initialization* lifecycle callback methods will be called on all objects regardless of scope, in the case of prototypes, any configured *destruction* lifecycle callbacks will *not* be called. It is the responsibility of the client code to clean up prototype scoped objects and release any expensive resources that the prototype bean(s) are holding onto. (One possible way to get the Spring container to release resources used by prototype-scoped beans is through the use of a custom bean post-processor which would hold a reference to the beans that need to be cleaned up.)

In some respects, you can think of the Spring containers role when talking about a prototype-scoped bean as somewhat of a replacement for the Java `'new'` operator. All lifecycle aspects past that point have to be handled by the client. (The lifecycle of a bean in the Spring container is further described in the section entitled Section 3.5.1, “Lifecycle interfaces”.)

3.4.3. Singleton beans with prototype-bean dependencies

When using singleton-scoped beans that have dependencies on beans that are scoped as prototypes, please be aware that *dependencies are resolved at instantiation time*. This means that if you dependency inject a prototype-scoped bean into a singleton-scoped bean, a brand new prototype bean will be instantiated and then dependency injected into the singleton bean... *but that is all*. That exact same prototype instance will be the sole instance that is ever supplied to the singleton-scoped bean, which is fine if that is what you want.

However, sometimes what you actually want is for the singleton-scoped bean to be able to acquire a brand new instance of the prototype-scoped bean again and again and again at runtime. In that case it is no use just dependency injecting a prototype-scoped bean into your singleton bean, because as explained above, that only happens *once* when the Spring container is instantiating the singleton bean and resolving and injecting its

dependencies. If you are in the scenario where you need to get a brand new instance of a (prototype) bean again and again and again at runtime, you are referred to the section entitled Section 3.3.8, “Method Injection”

Backwards compatibility note: specifying the lifecycle scope in XML

If you are referencing the `'spring-beans.dtd'` DTD in a bean definition file(s), and you are being explicit about the lifecycle scope of your beans you must use the "`singleton`" attribute to express the lifecycle scope (remembering that the singleton lifecycle scope is the default). If you are referencing the `'spring-beans-2.0.dtd'` DTD or the Spring 2.0 XSD schema, then you will need to use the "`scope`" attribute (because the "`singleton`" attribute was removed from the definition of the new DTD and XSD files in favour of the "`scope`" attribute).

To be totally clear about this, this means that if you use the "`singleton`" attribute in an XML bean definition then you *must* be referencing the `'spring-beans.dtd'` DTD *in that file*. If you are using the "`scope`" attribute then you *must* be referencing either the `'spring-beans-2.0.dtd'` DTD or the `'spring-beans-2.0.xsd'` XSD *in that file*.

3.4.4. The other scopes

The other scopes, namely `request`, `session`, and `global session` are for use only in web-based applications (and can be used irrespective of which particular web application framework you are using, if indeed any). In the interest of keeping related concepts together in one place in the reference documentation, these scopes are described here.

Note

The scopes that are described in the following paragraphs are *only* available if you are using a web-aware Spring `ApplicationContext` implementation (such as `XmlWebApplicationContext`). If you try using these next scopes with regular Spring IoC containers such as the `XmlBeanFactory` or `ClassPathXmlApplicationContext`, you *will* get an `IllegalStateException` complaining about an unknown bean scope.

3.4.4.1. Initial web configuration

In order to support the scoping of beans at the `request`, `session`, and `global session` levels (web-scoped beans), some minor initial configuration is required before you can set about defining your bean definitions. Please note that this extra setup is *not* required if you just want to use the 'standard' scopes (namely singleton and prototype).

Now as things stand, there are a couple of ways to effect this initial setup depending on your particular Servlet environment...

If you are accessing scoped beans within Spring Web MVC, i.e. within a request that is processed by the Spring `DispatcherServlet`, or `DispatcherPortlet`, then no special setup is necessary: `DispatcherServlet` and `DispatcherPortlet` already expose all relevant state.

When using a Servlet 2.4+ web container, with requests processed outside of Spring's DispatcherServlet (e.g. when using JSF or Struts), you need to add the following `javax.servlet.ServletRequestListener` to the declarations in your web application's `'web.xml'` file.

```
<web-app>
  ...
  <listener>
```

```
    <listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
  </listener>
  ...
</web-app>
```

If you are using an older web container (Servlet 2.3), you will need to use the provided `javax.servlet.Filter` implementation. Find below a snippet of XML configuration that has to be included in the `'web.xml'` file of your web application if you want to have access to web-scoped beans in requests outside of Spring's DispatcherServlet on a Servlet 2.3 container. (The filter mapping depends on the surrounding web application configuration and so you will have to change it as appropriate.)

```
<web-app>
  ..
  <filter>
    <filter-name>requestContextFilter</filter-name>
    <filter-class>org.springframework.web.filter.RequestContextFilter</filter-class>
  </filter>
  <filter-mapping>
    <filter-name>requestContextFilter</filter-name>
    <url-pattern>/*</url-pattern>
  </filter-mapping>
  ...
</web-app>
```

That's it. `DispatcherServlet`, `RequestContextListener` and `RequestContextFilter` all do exactly the same thing, namely bind the HTTP request object to the `Thread` that is servicing that request. This makes beans that are request- and session-scoped available further down the call chain.

3.4.4.2. The request scope

Consider the following bean definition:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

With the above bean definition in place, the Spring container will create a brand new instance of the `LoginAction` bean using the `'loginAction'` bean definition for each and every HTTP request. That is, the `'loginAction'` bean will be effectively scoped at the HTTP request level. You can change or dirty the internal state of the instance that is created as much as you want, safe in the knowledge that other requests that are also using instances created off the back of the same `'loginAction'` bean definition will not be seeing these changes in state since they are particular to an individual request. When the request is finished processing, the bean that is scoped to the request will be discarded.

3.4.4.3. The session scope

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

With the above bean definition in place, the Spring container will create a brand new instance of the `UserPreferences` bean using the `'userPreferences'` bean definition for the lifetime of a single HTTP `Session`. In other words, the `'userPreferences'` bean will be effectively scoped at the HTTP `Session` level. Just like `request-scoped` beans, you can change the internal state of the instance that is created as much as you want, safe in the knowledge that other HTTP `Session` instances that are also using instances created off the back of the same `'userPreferences'` bean definition will not be seeing these changes in state since they are particular to an individual HTTP `Session`. When the HTTP `Session` is eventually discarded, the bean that is scoped to that particular HTTP `Session` will also be discarded.

3.4.4.4. The global session scope

Consider the following bean definition:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="globalSession"/>
```

The `global session` scope is similar to the standard HTTP `Session` scope (described immediately above), and really only makes sense in the context of portlet-based web applications. The portlet specification defines the notion of a global `Session` that is shared amongst all of the various portlets that make up a single portlet web application. Beans defined at the `global session` scope are scoped (or bound) to the lifetime of the global portlet `Session`.

Please note that if you are writing a standard Servlet-based web application and you define one or more beans as having `global session` scope, the standard HTTP `Session` scope will be used, and no error will be raised.

3.4.4.5. Scoped beans as dependencies

Being able to define a bean scoped to a HTTP request or `Session` (or indeed a custom scope of your own devising) is all very well, but one of the main value-adds of the Spring IoC container is that it manages not only the instantiation of your objects (beans), but also the wiring up of collaborators (or dependencies). If you want to inject a (for example) HTTP request scoped bean into another bean, you will need to inject an AOP proxy in place of the scoped bean. That is, you need to inject a proxy object that exposes the same public interface as the scoped object, but that is smart enough to be able to retrieve the real, target object from the relevant scope (for example a HTTP request) and delegate method calls onto the real object.

Note

You *do not* need to use the `<aop:scoped-proxy/>` in conjunction with beans that are scoped as `singletons` or `prototypes`. It is an error to try to create a scoped proxy for a singleton bean (and the resulting `BeanCreationException` will certainly set you straight in this regard).

Let's look at the configuration that is required to effect this; the configuration is not hugely complex (it takes just one line), but it is important to understand the “why” as well as the “how” behind it.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <!-- a HTTP Session-scoped bean exposed as a proxy -->
    <bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

          <!-- this next element effects the proxying of the surrounding bean -->
          <aop:scoped-proxy/>
    </bean>

    <!-- a singleton-scoped bean injected with a proxy to the above bean -->
    <bean id="userService" class="com.foo.SimpleUserService">

        <!-- a reference to the proxied 'userPreferences' bean -->
        <property name="userPreferences" ref="userPreferences"/>

    </bean>
</beans>
```

To create such a proxy, you need only to insert a child `<aop:scoped-proxy/>` element into a scoped bean

definition (you may also need the CGLIB library on your classpath so that the container can effect class-based proxying; you will also need to be using Appendix A, *XML Schema-based configuration*). So, just why do you need this `<aop:scoped-proxy/>` element in the definition of beans scoped at the `request`, `session`, `globalSession` and '*insert your custom scope here*' level? The reason is best explained by picking apart the following bean definition (please note that the following `'userPreferences'` bean definition as it stands is *incomplete*):

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

From the above configuration it is evident that the singleton bean `'userManager'` is being injected with a reference to the HTTP `Session`-scoped bean `'userPreferences'`. The salient point here is that the `'userManager'` bean is a singleton... it will be instantiated *exactly once* per container, and its dependencies (in this case only one, the `'userPreferences'` bean) will also only be injected (once!). This means that the `'userManager'` will (conceptually) only ever operate on the exact same `'userPreferences'` object, that is the one that it was originally injected with. This is *not* what you want when you inject a HTTP `Session`-scoped bean as a dependency into a collaborating object (typically). Rather, what we *do* want is a single `'userManager'` object, and then, for the lifetime of a HTTP `Session`, we want to see and use a `'userPreferences'` object that is specific to said HTTP `Session`.

Rather what you need then is to inject some sort of object that exposes the exact same public interface as the `UserPreferences` class (ideally an object that *is a* `UserPreferences` instance) and that is smart enough to be able to go off and fetch the `real` `UserPreferences` object from whatever underlying scoping mechanism we have chosen (HTTP request, `Session`, etc.). We can then safely inject this proxy object into the `'userManager'` bean, which will be blissfully unaware that the `UserPreferences` reference that it is holding onto is a proxy. In the case of this example, when a `UserManager` instance invokes a method on the dependency-injected `UserPreferences` object, it is really invoking a method on the proxy... the proxy will then go off and fetch the real `UserPreferences` object from (in this case) the HTTP `Session`, and delegate the method invocation onto the retrieved real `UserPreferences` object.

That is why you need the following, correct and complete, configuration when injecting `request-`, `session-`, and `globalSession-scoped` beans into collaborating objects:

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">
    <aop:scoped-proxy/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

3.4.4.5.1. Choosing the type of proxy created

By default, when the Spring container is creating a proxy for a bean that is marked up with the `<aop:scoped-proxy/>` element, *a CGLIB-based class proxy will be created*. This means that you need to have the CGLIB library on the classpath of your application.

Note: CGLIB proxies will only intercept public method calls! Do not call non-public methods on such a proxy; they will not be delegated to the scoped target object.

You can choose to have the Spring container create 'standard' JDK interface-based proxies for such scoped beans by specifying '`false`' for the value of the '`proxy-target-class`' attribute of the `<aop:scoped-proxy/>` element. Using JDK interface-based proxies does mean that you don't need any additional libraries on your

application's classpath to effect such proxying, but it does mean that the class of the scoped bean must implement at least one interface, and *all* of the collaborators into which the scoped bean is injected must be referencing the bean via one of its interfaces.

```
<!-- DefaultUserPreferences implements the UserPreferences interface -->
<bean id="userPreferences" class="com.foo.DefaultUserPreferences" scope="session">
    <aop:scoped-proxy proxy-target-class="false"/>
</bean>

<bean id="userManager" class="com.foo.UserManager">
    <property name="userPreferences" ref="userPreferences"/>
</bean>
```

The section entitled Section 6.6, “Proxying mechanisms” may also be of some interest with regard to understanding the nuances of choosing whether class-based or interface-based proxying is right for you.

3.4.5. Custom scopes

As of Spring 2.0, the bean scoping mechanism in Spring is extensible. This means that you are not limited to just the bean scopes that Spring provides out of the box; you can define your own scopes, or even redefine the existing scopes (although that last one would probably be considered bad practice - please note that you *cannot* override the built-in `singleton` and `prototype` scopes).

3.4.5.1. Creating your own custom scope

Scopes are defined by the `org.springframework.beans.factory.config.Scope` interface. This is the interface that you will need to implement in order to integrate your own custom scope(s) into the Spring container, and is described in detail below. You may wish to look at the `Scope` implementations that are supplied with the Spring Framework itself for an idea of how to go about implementing your own. The Scope JavaDoc explains the main class to implement when you need your own scope in more detail too.

The `Scope` interface has four methods dealing with getting objects from the scope, removing them from the scope and allowing them to be 'destroyed' if needed.

The first method should return the object from the underlying scope. The session scope implementation for example will return the session-scoped bean (and if it does not exist, return a new instance of the bean, after having bound it to the session for future reference).

```
Object get(String name, ObjectFactory objectFactory)
```

The second method should remove the object from the underlying scope. The session scope implementation for example, removes the session-scoped bean from the underlying session. The object should be returned (you are allowed to return null if the object with the specified name wasn't found)

```
Object remove(String name)
```

The third method is used to register callbacks the scope should execute when it is destroyed or when the specified object in the scope is destroyed. Please refer to the JavaDoc or a Spring scope implementation for more information on destruction callbacks.

```
void registerDestructionCallback(String name, Runnable destructionCallback)
```

The last method deals with obtaining the conversation identifier for the underlying scope. This identifier is different for each scope. For a session for example, this can be the session identifier.

```
String getConversationId()
```

SPR-2600 - TODO

3.4.5.2. Using a custom scope

After you have written and tested one or more custom `Scope` implementations, you then need to make the Spring container aware of your new scope(s). The central method to register a new `Scope` with the Spring container is declared on the `ConfigurableBeanFactory` interface (implemented by most of the concrete `BeanFactory` implementations that ship with Spring); this central method is displayed below:

```
void registerScope(String scopeName, Scope scope);
```

The first argument to the `registerScope(..)` method is the unique name associated with a scope; examples of such names in the Spring container itself are `'singleton'` and `'prototype'`. The second argument to the `registerScope(..)` method is an actual instance of the custom `Scope` implementation that you wish to register and use.

Let's assume that you have written your own custom `Scope` implementation, and you have registered it like so:

```
// note: the ThreadScope class does not ship with the Spring Framework
Scope customScope = new ThreadScope();
beanFactory.registerScope("thread", scope);
```

You can then create bean definitions that adhere to the scoping rules of your custom `Scope` like so:

```
<bean id="..." class="..." scope="thread"/>
```

If you have your own custom `Scope` implementation(s), you are not just limited to only programmatic registration of the custom scope(s). You can also do the `Scope` registration declaratively, using the `CustomScopeConfigurer` class.

The declarative registration of custom `Scope` implementations using the `CustomScopeConfigurer` class is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <bean class="org.springframework.beans.factory.config.CustomScopeConfigurer">
        <property name="scopes">
            <map>
                <entry key="thread">
                    <bean class="com.foo.ThreadScope"/>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="bar" class="x.y.Bar" scope="thread">
        <property name="name" value="Rick"/>
        <aop:scoped-proxy/>
    </bean>

    <bean id="foo" class="x.y.Foo">
        <property name="bar" ref="bar"/>
    </bean>
```

```
</beans>
```

3.5. Customizing the nature of a bean

3.5.1. Lifecycle interfaces

The Spring Framework provides several marker interfaces to change the behavior of your bean in the container; they include `InitializingBean` and `DisposableBean`. Implementing these interfaces will result in the container calling `afterPropertiesSet()` for the former and `destroy()` for the latter to allow the bean to perform certain actions upon initialization and destruction.

Internally, the Spring Framework uses `BeanPostProcessor` implementations 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 `BeanPostProcessor` yourself. More information about this can be found in the section entitled Section 3.7, “Container extension points”.

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

3.5.1.1. Initialization callbacks

Implementing the `org.springframework.beans.factory.InitializingBean` interface allows a bean to perform initialization work after all necessary properties on the bean are set by the container. The `InitializingBean` interface specifies exactly one method:

```
void afterPropertiesSet() throws Exception;
```

Generally, the use of the `InitializingBean` interface can be avoided (and is discouraged since it unnecessarily couples the code to Spring). A bean definition provides support for a generic initialization method to be specified. In the case of XML-based configuration metadata, this is done using the `'init-method'` attribute. For example, the following definition:

```
<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
    }
}
```

... but does not couple the code to Spring.

3.5.1.2. Destruction callbacks

Implementing the `org.springframework.beans.factory.DisposableBean` interface allows a bean to get a callback when the container containing it is destroyed. The `DisposableBean` interface specifies one method:

```
void destroy() throws Exception;
```

Generally, the use of the `DisposableBean` marker interface can be avoided (and is discouraged since it unnecessarily couples the code to Spring). A bean definition provides support for a generic destroy method to be specified. When using XML-based configuration metadata this is done via the `'destroy-method'` attribute on the `<bean/>`. For example, the following definition:

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

```
public class ExampleBean {

    public void cleanup() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

Is exactly the same as...

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

```
public class AnotherExampleBean implements DisposableBean {

    public void destroy() {
        // do some destruction work (like releasing pooled connections)
    }
}
```

... but does not couple the code to Spring.

3.5.1.2.1. Default initialization & destroy methods

When you are writing initialization and destroy method callbacks that do not use the Spring-specific `InitializingBean` and `DisposableBean` callback interfaces, one (in the experience of this author) typically finds oneself writing methods with names such as `init()`, `initialize()`, `dispose()`, etc. The names of such lifecycle callback methods are (hopefully!) standardized across a project so that developers on a team all use the same method names and thus ensure some level of consistency.

The Spring container can now be configured to `'look'` for named initialization and destroy callback method names on *every* bean. This means that you as an application developer can simply write your application classes, use a convention of having an initialization callback called `init()`, and then (without having to configure each and every bean with, in the case of XML-based configuration, an `'init-method="init"'` attribute) be safe in the knowledge that the Spring IoC container *will* call that method when the bean is being created (and in accordance with the standard lifecycle callback contract described previously).

Let's look at an example to make the use of this feature completely clear. For the sake of the example, let us say that one of the coding conventions on a project is that all initialization callback methods are to be named `init()` and that destroy callback methods are to be called `destroy()`. This leads to classes like so...

```
public class DefaultBlogService implements BlogService {

    private BlogDao blogDao;

    public void setBlogDao(BlogDao blogDao) {
        this.blogDao = blogDao;
    }

    // this is (unsurprisingly) the initialization callback method
    public void init() {
        if (this.blogDao == null) {
            throw new IllegalStateException("The [blogDao] property must be set.");
        }
    }
}
```

```
<beans default-init-method="init">

    <bean id="blogService" class="com.foo.DefaultBlogService">
        <property name="blogDao" ref="blogDao" />
    </bean>

</beans>
```

Notice the use of the `'default-init-method'` attribute on the top-level `<beans/>` element. The presence of this attribute means that the Spring IoC container will recognize a method called `'init'` on beans as being the initialization method callback, and when a bean is being created and assembled, if the bean's class has such a method, it will be invoked at the appropriate time.

Destroy method callbacks are configured similarly (in XML that is) using the `'default-destroy-method'` attribute on the top-level `<beans/>` element.

The use of this feature can save you the (small) housekeeping chore of specifying an initialization and destroy method callback on each and every bean, and it is great for enforcing a consistent naming convention for initialization and destroy method callbacks (and consistency is something that should always be aimed for).

Consider the case where you have some existing beans where the underlying classes already have initialization callback methods that are named at variance with the convention. You can *always* override the default by specifying (in XML that is) the method name using the `'init-method'` and `'destroy-method'` attributes on the `<bean/>` element itself.

Finally, please be aware that the Spring container guarantees that a configured initialization callback is called immediately after a bean has been supplied with all of it's dependencies. This means that the initialization callback will be called on the raw bean reference, which means that any AOP interceptors or suchlike that will ultimately be applied to the bean will not yet be in place. A target bean is fully created *first*, *then* an AOP proxy (for example) with its interceptor chain is applied. Note that, if the target bean and the proxy are defined separately, your code can even interact to the raw target bean, bypassing the proxy. Hence, it would be very inconsistent to apply the interceptors to the init method, since that would couple the lifecycle of the target bean with its proxy/interceptors, and leave strange semantics when talking to the raw target bean directly.

3.5.1.2.2. Shutting down the Spring IoC container gracefully in non-web applications

Note

This next section does not apply to web applications (in case the title of this section did not make that abundantly clear). Spring's web-based `ApplicationContext` implementations already have code in place to handle shutting down the Spring IoC container gracefully when the relevant web application is being shutdown.

If you are using Spring's IoC container in a non-web application environment, for example in a rich client desktop environment, and you want the container to shutdown gracefully and call the relevant destroy callbacks on your singleton beans, you will need to register a shutdown hook with the JVM. This is quite easy to do (see below), and will ensure that your Spring IoC container shuts down gracefully and that all resources held by your singletons are released (of course it is still up to you to both configure the destroy callbacks for your singletons and implement such destroy callbacks correctly).

So to register a shutdown hook that enables the graceful shutdown of the relevant Spring IoC container, you simply need to call the `registerShutdownHook()` method that is declared on the `AbstractApplicationContext` class. To wit...

```
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        AbstractApplicationContext ctx
            = new ClassPathXmlApplicationContext(new String []{"beans.xml"});

        // add a shutdown hook for the above context...
        ctx.registerShutdownHook();

        // app runs here...

        // main method exits, hook is called prior to the app shutting down...
    }
}
```

3.5.2. Knowing who you are

3.5.2.1. `BeanFactoryAware`

A class which implements the `org.springframework.beans.factory.BeanFactoryAware` interface is provided with a reference to the `BeanFactory` that created it, when it is created by that `BeanFactory`.

```
public interface BeanFactoryAware {

    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

This allows beans to manipulate the `BeanFactory` that created them programmatically, through the `BeanFactory` interface, or by casting the reference to a known subclass of this which exposes additional functionality. Primarily this would consist of programmatic retrieval of other beans. While there are cases when this capability is useful, it should generally be avoided, since it couples the code to Spring, and does not follow the Inversion of Control style, where collaborators are provided to beans as properties.

An alternative option that is equivalent in effect to the `BeanFactoryAware`-based approach is to use the `org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean`. (It should be noted that this approach still does not reduce the coupling to Spring, but it does not violate the central principle of IoC as much as the `BeanFactoryAware`-based approach.)

The `ObjectFactoryCreatingFactoryBean` is a `FactoryBean` implementation that returns a reference to an object (factory) that can in turn be used to effect a bean lookup. The `ObjectFactoryCreatingFactoryBean` class does itself implement the `BeanFactoryAware` interface; what client beans are actually injected with is an instance of the `ObjectFactory` interface. This is a Spring-specific interface (and hence there is still no total decoupling from Spring), but clients can then use the `ObjectFactory`'s `getObject()` method to effect the bean lookup (under the hood the `ObjectFactory` implementation instance that is returned simply delegates down to a

`BeanFactory` to actually lookup a bean by name). All that you need to do is supply the `ObjectFactoryCreatingFactoryBean` with the name of the bean that is to be looked up. Let's look at an example:

```
package x.y;

public class NewsFeed {

    private String news;

    public void setNews(String news) {
        this.news = news;
    }

    public String getNews() {
        return this.toString() + ": '" + news + "'";
    }
}
```

```
package x.y;

import org.springframework.beans.factory.ObjectFactory;

public class NewsFeedManager {

    private ObjectFactory factory;

    public void setFactory(ObjectFactory factory) {
        this.factory = factory;
    }

    public void printNews() {
        // here is where the lookup is performed; note that there is no
        // need to hardcode the name of the bean that is being looked up...
        NewsFeed news = (NewsFeed) factory.getObject();
        System.out.println(news.getNews());
    }
}
```

Find below the XML configuration to wire together the above classes using the `ObjectFactoryCreatingFactoryBean` approach.

```
<beans>
    <bean id="newsFeedManager" class="x.y.NewsFeedManager">
        <property name="factory">
            <bean
class="org.springframework.beans.factory.config.ObjectFactoryCreatingFactoryBean">
                <property name="targetBeanName">
                    <idref local="newsFeed" />
                </property>
            </bean>
        </property>
    </bean>
    <bean id="newsFeed" class="x.y.NewsFeed" scope="prototype">
        <property name="news" value="... that's fit to print!" />
    </bean>
</beans>
```

And here is a small driver program to test the fact that new (prototype) instances of the `newsFeed` bean are actually being returned for each call to the injected `ObjectFactory` inside the `NewsFeedManager`'s `printNews()` method.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.NewsFeedManager;

public class Main {

    public static void main(String[] args) throws Exception {

```

```
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        NewsFeedManager manager = (NewsFeedManager) ctx.getBean("newsFeedManager");
        manager.printNews();
        manager.printNews();
    }
}
```

The output from running the above program will look like so (results will of course vary on your machine).

```
x.y.NewsFeed@1292d26: '... that's fit to print!'
x.y.NewsFeed@5329c5: '... that's fit to print!'
```

3.5.2.2. `BeanNameAware`

If a bean implements the `org.springframework.beans.factory.BeanNameAware` interface and is deployed in a `BeanFactory`, the `BeanFactory` will call the bean through this interface to inform the bean of the *id* it was deployed under. The callback will be invoked after population of normal bean properties but before an initialization callback like `InitializingBean`'s *afterPropertiesSet* or a custom init-method.

3.6. Bean definition inheritance

A bean definition potentially contains a large amount of configuration information, including container specific information (for example initialization method, static factory method name, and so forth) and constructor arguments and property values. A child bean definition is a bean definition that inherits configuration data from a parent definition. It is then able to override some values, or add others, as needed. Using parent and child bean definitions can potentially save a lot of typing. Effectively, this is a form of templating.

When working with a `BeanFactory` programmatically, child bean definitions are represented by the `ChildBeanDefinition` class. Most users will never work with them on this level, instead configuring bean definitions declaratively in something like the `XmlBeanFactory`. When using XML-based configuration metadata a child bean definition is indicated simply by using the `'parent'` attribute, specifying the parent bean as the value of this attribute.

```
<bean id="inheritedTestBean" abstract="true"
    class="org.springframework.beans.TestBean">
  <property name="name" value="parent"/>
  <property name="age" value="1"/>
</bean>

<bean id="inheritsWithDifferentClass"
      class="org.springframework.beans.DerivedTestBean"
      parent="inheritedTestBean" init-method="initialize">

  <property name="name" value="override"/>
  <!-- the age property value of 1 will be inherited from  parent -->

</bean>
```

A child bean definition will use the bean class from the parent definition if none is specified, but can also override it. In the latter case, the child bean class must be compatible with the parent, that is it must accept the parent's property values.

A child bean definition will inherit constructor argument values, property values and method overrides from the parent, with the option to add new values. If any init-method, destroy-method and/or `static` factory method settings are specified, they will override the corresponding parent settings.

The remaining settings will *always* be taken from the child definition: *depends on*, *autowire mode*, *dependency*

check, singleton, scope, lazy init.

Note that in the example above, we have explicitly marked the parent bean definition as abstract by using the `abstract` attribute. In the case that the parent definition does not specify a class, and so explicitly marking the parent bean definition as `abstract` is required:

```
<bean id="inheritedTestBeanWithoutClass" abstract="true">
    <property name="name" value="parent"/>
    <property name="age" value="1"/>
</bean>

<bean id="inheritsWithClass" class="org.springframework.beans.DerivedTestBean"
    parent="inheritedTestBeanWithoutClass" init-method="initialize">
  <property name="name" value="override"/>
  <!-- age will inherit the value of 1 from the parent bean definition-->
</bean>
```

The parent bean cannot get instantiated on its own since it is incomplete, and it is also explicitly marked as `abstract`. When a definition is defined to be `abstract` like this, it is usable only as a pure template bean definition that will serve as a parent definition for child definitions. Trying to use such an `abstract` parent bean on its own (by referring to it as a ref property of another bean, or doing an explicit `getBean()` call with the parent bean id), will result in an error. Similarly, the container's internal `preInstantiateSingletons()` method will completely ignore bean definitions which are defined as abstract.

Note

`ApplicationContexts` (but *not* `BeanFactories`) will by default pre-instantiate all singletons. Therefore it is important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *'abstract'* attribute to *'true'*, otherwise the application context will actually (attempt to) pre-instantiate the `abstract` bean.

3.7. Container extension points

The IoC component of the Spring Framework has been designed for extension. There is typically no need for an application developer to subclass any of the various `BeanFactory` or `ApplicationContext` implementation classes. The Spring IoC container can be infinitely extended by plugging in implementations of special integration interfaces. The next few sections are devoted to detailing all of these various integration interfaces.

3.7.1. Customizing beans using `BeanPostProcessors`

The first extension point that we will look at is the `BeanPostProcessor` interface. This interface defines a number of *callback methods* that you as an application developer can implement in order to provide your own (or override the containers default) instantiation logic, dependency-resolution logic, and so forth. If you want to do some custom logic after the Spring container has finished instantiating, configuring and otherwise initializing a bean, you can plug in one or more `BeanPostProcessor` implementations.

You can configure multiple `BeanPostProcessors` if you wish. You can control the order in which these `BeanPostProcessors` execute by setting the `'order'` property (you can only set this property if the `BeanPostProcessor` implements the `Ordered` interface; if you write your own `BeanPostProcessor` you should consider implementing the `Ordered` interface too); consult the Javadocs for the `BeanPostProcessor` and `Ordered` interfaces for more details.

Note

`BeanPostProcessors` operate on bean (or object) *instances*; that is to say, the Spring IoC container will have instantiated a bean instance for you, and *then* `BeanPostProcessors` get a chance to do their stuff.

If you want to change the actual bean definition (that is the recipe that defines the bean), then you rather need to use a `BeanFactoryPostProcessor` (described below in the section entitled Section 3.7.2, “Customizing configuration metadata with `BeanFactoryPostProcessors`”.

Also, `BeanPostProcessors` are scoped *per-container*. This is only relevant if you are using container hierarchies. If you define a `BeanPostProcessor` in one container, it will *only* do its stuff on the beans in that container. Beans that are defined in another container will not be post-processed by `BeanPostProcessors` in another container, even if both containers are part of the same hierarchy.

The `org.springframework.beans.factory.config.BeanPostProcessor` interface consists of exactly two callback methods. When such a class is registered as a post-processor with the container (see below for how this registration is effected), for each bean instance that is created by the container, the post-processor will get a callback from the container both *before* any container initialization methods (such as *afterPropertiesSet* and any declared init method) are called, and also afterwards. The post-processor is free to do what it wishes with the bean instance, including ignoring the callback completely. A bean post-processor will typically check for marker interfaces, or do something such as wrap a bean with a proxy; some of the Spring AOP infrastructure classes are implemented as bean post-processors and they do this proxy-wrapping logic.

It is important to know that a `BeanFactory` treats bean post-processors slightly differently than an `ApplicationContext`. An `ApplicationContext` will *automatically detect* any beans which are defined in the configuration metadata which is supplied to it that implement the `BeanPostProcessor` interface, and register them as post-processors, to be then called appropriately by the container on bean creation. Nothing else needs to be done other than deploying the post-processors in a similar fashion to any other bean. On the other hand, when using a `BeanFactory` implementation, bean post-processors explicitly have to be registered, with code like this:

```
ConfigurableBeanFactory factory = new XmlBeanFactory(...);

// now register any needed BeanPostProcessor instances
MyBeanPostProcessor postProcessor = new MyBeanPostProcessor();
factory.addBeanPostProcessor(postProcessor);

// now start using the factory
```

This explicit registration step is not convenient, and this is one of the reasons why the various `ApplicationContext` implementations are preferred above plain `BeanFactory` implementations in the vast majority of Spring-backed applications, especially when using `BeanPostProcessors`.

`BeanPostProcessors` and AOP auto-proxying

Classes that implement the `BeanPostProcessor` interface are *special*, and so they are treated differently by the container. All `BeanPostProcessors` *and their directly referenced beans* will be instantiated on startup, as part of the special startup phase of the `ApplicationContext`, *then* all those `BeanPostProcessors` will be registered in a sorted fashion - and applied to all further beans. Since AOP auto-proxying is implemented as a `BeanPostProcessor` itself, no `BeanPostProcessors` or directly referenced beans are eligible for auto-proxying (and thus will not have aspects 'woven' into them.

> For any such bean, you should see an info log message: "*Bean 'foo' is not eligible for getting processed by all BeanPostProcessors (for example: not eligible for auto-proxying)*".

Find below some examples of how to write, register, and use `BeanPostProcessors` in the context of an `ApplicationContext`.

3.7.1.1. Example: Hello World, `BeanPostProcessor`-style

This first example is hardly compelling, but serves to illustrate basic usage. All we are going to do is code a custom `BeanPostProcessor` implementation that simply invokes the `toString()` method of each bean as it is created by the container and prints the resulting string to the system console. Yes, it is not hugely useful, but serves to get the basic concepts across before we move into the second example which *is* actually useful.

Find below the custom `BeanPostProcessor` implementation class definition:

```
package scripting;

import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.beans.BeansException;

public class InstantiationTracingBeanPostProcessor implements BeanPostProcessor {

    // simply return the instantiated bean as-is
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
        return bean; // we could potentially return any object reference here...
    }

    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {
        System.out.println("Bean '" + beanName + "' created : " + bean.toString());
        return bean;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

    <lang:groovy id="messenger"
          script-source="classpath:org/springframework/scripting/groovy/Messenger.groovy">
        <lang:property name="message" value="Fiona Apple Is Just So Dreamy."/>
    </lang:groovy>

    <!--
        when the above bean ('messenger') is instantiated, this custom
        BeanPostProcessor implementation will output the fact to the system console
     -->
    <bean class="scripting.InstantiationTracingBeanPostProcessor"/>

</beans>
```

Notice how the `InstantiationTracingBeanPostProcessor` is simply defined; it doesn't even have a name, and because it is a bean it can be dependency injected just like any other bean. (The above configuration also just so happens to define a bean that is backed by a Groovy script. The Spring 2.0 dynamic language support is detailed in the chapter entitled Chapter 24, *Dynamic language support*.)

Find below a small driver script to exercise the above code and configuration;

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("scripting/beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger);
    }
}
```

The output of executing the above program will be (something like) this:

```
Bean 'messenger' created : org.springframework.scripting.groovy.GroovyMessenger@272961
org.springframework.scripting.groovy.GroovyMessenger@272961
```

3.7.1.2. Example: The `RequiredAnnotationBeanPostProcessor`

Using marker interfaces or annotations in conjunction with a custom `BeanPostProcessor` implementation is a common means of extending the Spring IoC container. This next example is a bit of a cop-out, in that you are directed to the section entitled Section 25.3.1, "`@Required`" which demonstrates the usage of a custom `BeanPostProcessor` implementation that ships with the Spring distribution which ensures that JavaBean properties on beans that are marked with an (arbitrary) annotation are actually (configured to be) dependency-injected with a value.

3.7.2. Customizing configuration metadata with `BeanFactoryPostProcessors`

The next extension point that we will look at is the `org.springframework.beans.factory.config.BeanFactoryPostProcessor`. The semantics of this interface are similar to the `BeanPostProcessor`, with one major difference: `BeanFactoryPostProcessors` operate on the *bean configuration metadata*; that is, the Spring IoC container will allow `BeanFactoryPostProcessors` to read the configuration metadata and potentially change it *before* the container has actually instantiated any other beans.

You can configure multiple `BeanFactoryPostProcessors` if you wish. You can control the order in which these `BeanFactoryPostProcessors` execute by setting the `'order'` property (you can only set this property if the `BeanFactoryPostProcessor` implements the `Ordered` interface; if you write your own `BeanFactoryPostProcessor` you should consider implementing the `Ordered` interface too); consult the Javadocs for the `BeanFactoryPostProcessor` and `Ordered` interfaces for more details.

Note

If you want to change the actual bean *instances* (the objects that are created from the configuration metadata), then you rather need to use a `BeanPostProcessor` (described above in the section entitled Section 3.7.1, "Customizing beans using `BeanPostProcessors`".

Also, `BeanFactoryPostProcessors` are scoped *per-container*. This is only relevant if you are using container hierarchies. If you define a `BeanFactoryPostProcessor` in one container, it will *only* do its stuff on the bean definitions in that container. Bean definitions in another container will not be post-processed by `BeanFactoryPostProcessors` in another container, even if both containers are part of the same hierarchy.

A bean factory post-processor is executed manually (in the case of a `BeanFactory`) or automatically (in the case of an `ApplicationContext`) to apply changes of some sort to the configuration metadata that defines a

container. Spring includes a number of pre-existing bean factory post-processors, such as `PropertyResourceConfigurer` and `PropertyPlaceholderConfigurer`, both described below, and `BeanNameAutoProxyCreator`, which is very useful for wrapping other beans transactionally or with any other kind of proxy, as described later in this manual. The `BeanFactoryPostProcessor` can be used to add custom property editors.

In a `BeanFactory`, the process of applying a `BeanFactoryPostProcessor` is manual, and will be similar to this:

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

// bring in some property values from a Properties file
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));

// now actually do the replacement
cfg.postProcessBeanFactory(factory);
```

This explicit registration step is not convenient, and this is one of the reasons why the various `ApplicationContext` implementations are preferred above plain `BeanFactory` implementations in the vast majority of Spring-backed applications, especially when using `BeanFactoryPostProcessors`.

An `ApplicationContext` will detect any beans which are deployed into it which implement the `BeanFactoryPostProcessor` interface, and automatically use them as bean factory post-processors, at the appropriate time. Nothing else needs to be done other than deploying these post-processor in a similar fashion to any other bean.

Note

Just as in the case of `BeanPostProcessors`, you typically don't want to have `BeanFactoryPostProcessors` marked as being lazily-initialized. If they are marked as such, then the Spring container will never instantiate them, and thus they won't get a chance to apply their custom logic. If you are using the `'default-lazy-init'` attribute on the declaration of your `<beans/>` element, be sure to mark your various `BeanFactoryPostProcessor` bean definitions with `'lazy-init="false"'`.

3.7.2.1. Example: the `PropertyPlaceholderConfigurer`

The `PropertyPlaceholderConfigurer` is used to externalize property values from a `BeanFactory` definition, into another separate file in the standard Java `Properties` format. This is useful to allow the person deploying an application to customize enviroment-specific properties (for example database URLs, usernames and passwords), without the complexity or risk of modifying the main XML definition file or files for the container.

Consider the following XML-based configuration metadata fragment, where a `DataSource` with placeholder values is defined. We will configure some properties from an external `Properties` file, and at runtime, we will apply a `PropertyPlaceholderConfigurer` to the metadata which will replace some properties of the datasource:

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <value>classpath:com/foo/jdbc.properties</value>
    </property>
</bean>

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

The actual values come from another file in the standard Java `Properties` format:

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

The `PropertyPlaceholderConfigurer` doesn't only look for properties in the `Properties` file you specify, but also checks against the Java `System` properties if it cannot find a property you are trying to use. This behavior can be customized by setting 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 `PropertyPlaceholderConfigurer` for more information.

Class name substitution

The `PropertyPlaceholderConfigurer` can be used to substitute class names, which is sometimes useful when you have to pick a particular implementation class at runtime. For example:

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <value>classpath:com/foo/strategy.properties</value>
    </property>
    <property name="properties">
        <value>custom.strategy.class=com.foo.DefaultStrategy</value>
    </property>
</bean>

<bean id="serviceStrategy" class="${custom.strategy.class}"/>
```

If the class is unable to be resolved at runtime to a valid class, resolution of the bean will fail once it is about to be created (which is during the `preInstantiateSingletons()` phase of an `ApplicationContext` for a non-lazy-init bean.)

3.7.2.2. Example: the `PropertyOverrideConfigurer`

The `PropertyOverrideConfigurer`, another bean factory post-processor, is similar to the `PropertyPlaceholderConfigurer`, but in contrast to the latter, the original definitions can have default values or no values at all for bean properties. If an overriding `Properties` file does not have an entry for a certain bean property, the default context definition is used.

Note that the bean factory definition is *not* aware of being overridden, so it is not immediately obvious when looking at the XML definition file that the override configurer is being used. In case that there are multiple `PropertyOverrideConfigurer` instances that define different values for the same bean property, the last one will win (due to the overriding mechanism).

Properties file configuration lines are expected to be in the format:

```
beanName.property=value
```

An example properties file might look like this:

```
dataSource.driverClassName=com.mysql.jdbc.Driver
dataSource.url=jdbc:mysql:mydb
```

This example file would be usable against a container definition which contains a bean called *dataSource*, which has *driver* and *url* properties.

Note that compound property names are also supported, as long as every component of the path except the final property being overridden is already non-null (presumably initialized by the constructors). In this example...

```
foo.fred.bob.sammy=123
```

... the `sammy` property of the `bob` property of the `fred` property of the `foo` bean is being set to the scalar value `123`.

3.7.3. Customizing instantiation logic using `FactoryBeans`

The `org.springframework.beans.factory.FactoryBean` interface is to be implemented by objects that *are themselves factories*.

The `FactoryBean` interface is a point of pluggability into the Spring IoC containers instantiation logic. If you have some complex initialization code that is better expressed in Java as opposed to a (potentially) verbose amount of XML, you can create your own `FactoryBean`, write the complex initialization inside that class, and then plug your custom `FactoryBean` into the container.

The `FactoryBean` interface provides three methods:

- `Object getObject()`: has to return an instance of the object this factory creates. The instance can possibly be shared (depending on whether this factory returns singletons 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

The `FactoryBean` concept and interface is used in a number of places within the Spring Framework; at the time of writing there are over 50 implementations of the `FactoryBean` interface that ship with Spring itself.

Finally, there is sometimes a need to ask a container for an actual `FactoryBean` instance itself, not the bean it produces. This may be achieved by prepending the bean id with `'&'` (sans quotes) when calling the `getBean` method of the `BeanFactory` (including `ApplicationContext`). So for a given `FactoryBean` with an id of `myBean`, invoking `getBean("myBean")` on the container will return the product of the `FactoryBean`, but invoking `getBean("&myBean")` will return the `FactoryBean` instance itself.

3.8. The `ApplicationContext`

While the `beans` package provides basic functionality for managing and manipulating beans, often in a programmatic way, the `context` package adds `ApplicationContext`, which enhances `BeanFactory` functionality in a more *framework-oriented style*. Many users will use `ApplicationContext` in a completely declarative fashion, not even having to create it manually, but instead relying on support classes such as `ContextLoader` to automatically start an `ApplicationContext` as part of the normal startup process of a J2EE web-app. Of course, it is still possible to programmatically create an ApplicationContext.

The basis for the context package is the `ApplicationContext` interface, located in the `org.springframework.context` package. Deriving from the `BeanFactory` interface, it provides all the

functionality of `BeanFactory`. To allow working in a more framework-oriented fashion, using layering and hierarchical contexts, the context package also provides the following functionality:

- `MessageSource`, providing access to messages in i18n-style
- *Access to resources*, such as URLs and files
- *Event propagation* to beans implementing the `ApplicationListener` interface
- *Loading of multiple (hierarchical) contexts*, allowing each to be focused on one particular layer, for example the web layer of an application

As the `ApplicationContext` includes all functionality of the `BeanFactory`, it is generally recommended that it be used over the `BeanFactory`, except for a few limited situations such as perhaps in an `Applet`, where memory consumption might be critical, and a few extra kilobytes might make a difference. The following sections describe functionality that `ApplicationContext` adds to basic `BeanFactory` capabilities.

3.8.1. Internationalization using `MessageSources`

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

- `String getMessage(String code, Object[] args, String default, Locale loc)`: the basic method used to retrieve a message from the `MessageSource`. When no message is found for the specified locale, the default message is used. Any arguments passed in are used as replacement values, using the `MessageFormat` functionality provided by the standard library.
- `String getMessage(String code, Object[] args, Locale loc)`: essentially the same as the previous method, but with one difference: no default message can be specified; if the message cannot be found, a `NoSuchMessageException` is thrown.
- `String getMessage(MessageSourceResolvable resolvable, Locale locale)`: all properties used in the methods above are also wrapped in a class named `MessageSourceResolvable`, which you can use via 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 described above will be delegated to the message source that was found. If no message source was found, the `ApplicationContext` attempts to see if it has a parent containing a bean with the same 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 currently provides two `MessageSource` implementations. These are the `ResourceBundleMessageSource` and the `StaticMessageSource`. Both implement `HierarchicalMessageSource` in order to do nested messaging. The `StaticMessageSource` is hardly ever used but provides programmatic ways to add messages to the source. The `ResourceBundleMessageSource` is more interesting and is the one we will provide 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. For the purposes of the example, lets assume the contents of two of the above resource bundle files are...

```
# in 'format.properties'
message=Alligators rock!
```

```
# in 'exceptions.properties'
argument.required=The '{0}' argument is required.
```

Some (admittedly trivial) driver code to exercise the `MessageSource` functionality can be found below. Remember that all `ApplicationContext` implementations are also `MessageSource` implementations and so can be cast to the `MessageSource` interface.

```
public static void main(String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("message", null, "Default", null);
    System.out.println(message);
}
```

The resulting output from the above program will be...

```
Alligators rock!
```

So to summarize, the `MessageSource` is defined in a file called `'beans.xml'` (this file exists at the root of your classpath). The `'messageSource'` bean definition refers to a number of resource bundles via it's `basenames` property; the three files that are passed in the list to the `basenames` property exist as files at the root of your classpath (and are called `format.properties`, `exceptions.properties`, and `windows.properties` respectively).

Lets look at another example, and this time we will look at passing arguments to the message lookup; these arguments will be converted into strings and inserted into placeholders in the lookup message. This is perhaps best explained with an example:

```
<beans>

    <!-- this MessageSource is being used in a web application -->
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="baseName" value="WEB-INF/test-messages"/>
    </bean>

    <!-- let's inject the above MessageSource into this POJO -->
    <bean id="example" class="com.foo.Example">
        <property name="messages" ref="messageSource"/>
    </bean>

</beans>
```

```
public class Example {

    private MessageSource messages;

    public void setMessages(MessageSource messages) {
```

```
        this.messages = messages;
    }

    public void execute() {
        String message = this.messages.getMessage("argument.required",
            new Object [] {"userDao"}, "Required", null);
        System.out.println(message);
    }

}
```

The resulting output from the invocation of the `execute()` method will be...

```
The 'userDao' argument is required.
```

With regard to internationalization (i18n), Spring's various `MessageResource` implementations follow the same locale resolution and fallback rules as the standard JDK `ResourceBundle`. In short, and continuing with the example `'messageSource'` defined previously, if you want to resolve messages against the British (en-GB) locale, you would create files called `format_en_GB.properties`, `exceptions_en_GB.properties`, and `windows_en_GB.properties` respectively.

Locale resolution is typically going to be managed by the surrounding environment of the application. For the purpose of this example though, we'll just manually specify the locale that we want to resolve our (British) messages against.

```
# in 'exceptions_en_GB.properties'
argument.required=Ebagum lad, the '{0}' argument is required, I say, required.
```

```
public static void main(final String[] args) {
    MessageSource resources = new ClassPathXmlApplicationContext("beans.xml");
    String message = resources.getMessage("argument.required",
        new Object [] {"userDao"}, "Required", Locale.UK);
    System.out.println(message);
}
```

The resulting output from the running of the above program will be...

```
Ebagum lad, the 'userDao' argument is required, I say, required.
```

The `MessageSourceAware` interface can also be used to acquire a reference to any `MessageSource` that has been defined. Any bean that is defined in an `ApplicationContext` that implements the `MessageSourceAware` interface will be injected with the application context's `MessageSource` when it (the bean) is being created and configured.

3.8.2. Events

Event handling in the `ApplicationContext` is provided through the `ApplicationEvent` class and `ApplicationListener` interface. If a bean which implements the `ApplicationListener` interface is deployed into the context, every time an `ApplicationEvent` gets published to the `ApplicationContext`, that bean will be notified. Essentially, this is the standard *Observer* design pattern. Spring provides three standard events:

Table 3.5. Built-in Events

Event	Explanation
`ContextRefreshedEvent`	

Event	Explanation
	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`	Published when the `ApplicationContext` is closed, using the `close()` method on the `ApplicationContext`. Closed here means that singleton beans (only!) are destroyed.
`RequestHandledEvent`	A web-specific event telling all beans that a HTTP request has been serviced (this will 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. Simply call the `publishEvent()` method on the `ApplicationContext`, specifying a parameter which is an instance of your custom event class implementing `ApplicationEvent`. Event listeners receive events synchronously. This means the `publishEvent()` method blocks until all listeners have finished processing the event (it is possible to supply an alternate event publishing strategy via a `ApplicationEventMulticaster` implementation). Furthermore, when a listener receives an event it operates inside the transaction context of the publisher, if a transaction context is available.

Let's 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"/>
</bean>
```

Now, let's look at the actual classes:

```
public class EmailBean implements ApplicationContextAware {

    private List blackList;
        private ApplicationContext ctx;

    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 implements ApplicationListener {

    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 particular example could probably be implemented in better ways (perhaps by using AOP features), but it should be sufficient to illustrate the basic event mechanism.

3.8.3. Convenient access to low-level resources

For optimal usage and understanding of application contexts, users should generally familiarize themselves with Spring's `Resource` abstraction, as described in the chapter entitled Chapter 4, *Resources*.

An application context is a `ResourceLoader`, able to be used to load `Resources`. A `Resource` is essentially a `java.net.URL` on steroids (in fact, it just wraps and uses a URL where appropriate), which can be used to obtain low-level resources from almost any location in a transparent fashion, including from the classpath, a filesystem location, anywhere describable with a standard URL, and some other variations. If the resource location string is a simple path without any special prefixes, where those resources come from is specific and appropriate to the actual application context type.

A bean deployed into the application context may implement the special marker interface, `ResourceLoaderAware`, to be automatically called back at initialization time with the application context itself passed in as the `ResourceLoader`. A bean may also expose properties of type `Resource`, to be used to access static resources, and expect that they will be injected into it like any other properties. The person deploying the bean may specify those `Resource` properties as simple String paths, and rely on a special JavaBean `PropertyEditor` that is automatically registered by the context, to convert those text strings to actual `Resource` objects.

The location path or paths supplied to an `ApplicationContext` constructor are actually resource strings, and in simple form are treated appropriately to the specific context implementation (`ClassPathXmlApplicationContext` treats a simple location path as a classpath location), but may also be used with special prefixes to force loading of definitions from the classpath or a URL, regardless of the actual context type.

3.8.4. Convenient `ApplicationContext` instantiation for web applications

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

The `ContextLoader` mechanism comes in two flavors: the `ContextLoaderListener` and the `ContextLoaderServlet`. They both have the same functionality but differ in that the listener version cannot be reliably used in Servlet 2.3 containers. Since the Servlet 2.4 specification, servlet context listeners are required to execute immediately after the servlet context for the web application has been created and is available to service the first request (and also when the servlet context is about to be shut down): as such a servlet context

listener is an ideal place to initialize the Spring `ApplicationContext`. 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 above 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 the parameter does not exist, the listener will use `/WEB-INF/applicationContext.xml` as a default. When it *does* exist, it will separate the String using predefined delimiters (comma, semi-colon and whitespace) and use the values as locations where application contexts will be searched for. Ant-style path patterns are supported as well: e.g. `/WEB-INF/*Context.xml` (for all files whose name ends with "Context.xml", residing in the "WEB-INF" directory) or `/WEB-INF/**/*Context.xml` (for all such files in any subdirectory of "WEB-INF").

The `ContextLoaderServlet` can be used instead of the `ContextLoaderListener`. The servlet will use the `'contextConfigLocation'` parameter just as the listener does.

3.9. Glue code and the evil singleton

The majority of the code inside an application is best written in a DI style, where that code is served out of a Spring IoC 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 Spring IoC container. 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 Spring IoC container. 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 Spring IoC container to get a real object to delegate to, then inversion of control has still been achieved for the majority of the code (the object coming out of the container); 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 Spring IoC container. While the Spring IoC container itself 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 Spring IoC container such as a Hibernate `SessionFactory`) for each bean to use its own, non-singleton Spring IoC container.

As another example, in complex J2EE apps with multiple layers (various JAR files, EJBs, and WAR files packaged as an EAR), with each layer having its own Spring IoC container 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 Spring IoC container from the multiple XML definition files from each layer. All of the various Spring IoC container implementations may be constructed from multiple definition files in this fashion. However, if there are multiple sibling web-applications at the root of the hierarchy, it is problematic to create a Spring IoC container for each web-application which consists of mostly identical bean definitions from lower

layers, as there may be issues due to increased memory usage, issues with creating multiple copies of beans which take a long time to initialize (for example a Hibernate `SessionFactory`), and possible issues due to side-effects. As an alternative, classes such as `ContextSingletonBeanFactoryLocator` or `SingletonBeanFactoryLocator` may be used to demand-load multiple hierarchical (that is one container is the parent of another) Spring IoC container instances in a singleton fashion, which may then be used as the parents of the web-application Spring IoC container instances. The result is that bean definitions for lower layers are loaded only as needed, and loaded only once.

3.9.1. Using the Singleton-helper classes

You can see a detailed example of their usage in SingletonBeanFactoryLocator and ContextSingletonBeanFactoryLocator by viewing their respective Javadocs.

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. Resources

4.1. Introduction

Java's standard `java.net.URL` class and standard handlers for various URL prefixes unfortunately are not quite adequate enough for all access to low-level resources. For example, there is no standardized `URL` implementation that may be used to access a resource that needs to be obtained from the classpath, or relative to a `ServletContext`. While it is possible to register new handlers for specialized `URL` prefixes (similar to existing handlers for prefixes such as `http:`), this is generally quite complicated, and the `URL` interface still lacks some desirable functionality, such as a method to check for the existence of the resource being pointed to.

4.2. The `Resource` interface

Spring's `Resource` interface is meant to be a more capable interface for abstracting access to low-level resources.

```
public interface Resource extends InputStreamSource {

    boolean exists();

    boolean isOpen();

    URL getURL() throws IOException;

    File getFile() throws IOException;

    Resource createRelative(String relativePath) throws IOException;

    String getFilename();

    String getDescription();
}
```

```
public interface InputStreamSource {

    InputStream getInputStream() throws IOException;
}
```

Some of the most important methods from the `Resource` interface are:

- `getInputStream()`: locates and opens the resource, returning an `InputStream` for reading from the resource. It is expected that each invocation returns a fresh `InputStream`. It is the responsibility of the caller to close the stream.
- `exists()`: returns a `boolean` indicating whether this resource actually exists in physical form.
- `isOpen()`: returns a `boolean` indicating whether this resource represents a handle with an open stream. If `true`, the `InputStream` cannot be read multiple times, and must be read once only and then closed to avoid resource leaks. Will be `false` for all usual resource implementations, with the exception of `InputStreamResource`.
- `getDescription()`: returns a description for this resource, to be used for error output when working with the resource. This is often the fully qualified file name or the actual URL of the resource.

Other methods allow you to obtain an actual `URL` or `File` object representing the resource (if the underlying

implementation is compatible, and supports that functionality).

The `Resource` abstraction is used extensively in Spring itself, as an argument type in many method signatures when a resource is needed. Other methods in some Spring APIs (such as the constructors to various `ApplicationContext` implementations), take a `String` which in unadorned or simple form is used to create a `Resource` appropriate to that context implementation, or via special prefixes on the `String` path, allow the caller to specify that a specific `Resource` implementation must be created and used.

While the `Resource` interface is used a lot with Spring and by Spring, it's actually very useful to use as a general utility class by itself in your own code, for access to resources, even when your code doesn't know or care about any other parts of Spring. While this couples your code to Spring, it really only couples it to this small set of utility classes, which are serving as a more capable replacement for `URL`, and can be considered equivalent to any other library you would use for this purpose.

It is important to note that the `Resource` abstraction does not replace functionality: it wraps it where possible. For example, a `UrlResource` wraps a URL, and uses the wrapped `URL` to do it's work.

4.3. Built-in `Resource` implementations

There are a number of `Resource` implementations that come supplied straight out of the box in Spring:

4.3.1. `UrlResource`

The `UrlResource` wraps a `java.net.URL`, and may be used to access any object that is normally accessible via a URL, such as files, an HTTP target, an FTP target, etc. All URLs have a standardized `String` representation, such that appropriate standardized prefixes are used to indicate one URL type from another. This includes `file:` for accessing filesystem paths, `http:` for accessing resources via the HTTP protocol, `ftp:` for accessing resources via FTP, etc.

A `UrlResource` is created by Java code explicitly using the `UrlResource` constructor, but will often be created implicitly when you call an API method which takes a `String` argument which is meant to represent a path. For the latter case, a JavaBeans `PropertyEditor` will ultimately decide which type of `Resource` to create. If the path string contains a few well-known (to it, that is) prefixes such as `classpath:`, it will create an appropriate specialized `Resource` for that prefix. However, if it doesn't recognize the prefix, it will assume the this is just a standard URL string, and will create a `UrlResource`.

4.3.2. `ClassPathResource`

This class represents a resource which should be obtained from the classpath. This uses either the thread context class loader, a given class loader, or a given class for loading resources.

This `Resource` implementation supports resolution as `java.io.File` if the class path resource resides in the file system, but not for classpath resources which reside in a jar and have not been expanded (by the servlet engine, or whatever the environment is) to the filesystem. To address this the various `Resource` implementations always support resolution as a `java.net.URL`.

A `ClassPathResource` is created by Java code explicitly using the `ClassPathResource` constructor, but will often be created implicitly when you call an API method which takes a `String` argument which is meant to represent a path. For the latter case, a JavaBeans `PropertyEditor` will recognize the special prefix `classpath:`on the string path, and create a `ClassPathResource` in that case.

4.3.3. `FileSystemResource`

This is a `Resource` implementation for `java.io.File` handles. It obviously supports resolution as a `File`, and as a `URL`.

4.3.4. `ServletContextResource`

This is a `Resource` implementation for `ServletContext` resources, interpreting relative paths within the relevant web application's root directory.

This always supports stream access and URL access, but only allows `java.io.File` access when the web application archive is expanded and the resource is physically on the filesystem. Whether or not it's expanded and on the filesystem like this, or accessed directly from the JAR or somewhere else like a DB (it's conceivable) is actually dependent on the Servlet container.

4.3.5. `InputStreamResource`

A `Resource` implementation for a given `InputStream`. This should only be used if no specific `Resource` implementation is applicable. In particular, prefer `ByteArrayResource` or any of the file-based `Resource` implementations where possible.

In contrast to other `Resource` implementations, this is a descriptor for an *already* opened resource - therefore returning `true` from `isOpen()`. Do not use it if you need to keep the resource descriptor somewhere, or if you need to read a stream multiple times.

4.3.6. `ByteArrayResource`

This is a `Resource` implementation for a given byte array. It creates a `ByteArrayInputStream` for the given byte array.

It's useful for loading content from any given byte array, without having to resort to a single-use `InputStreamResource`.

4.4. The `ResourceLoader`

The `ResourceLoader` interface is meant to be implemented by objects that can return (i.e. load) `Resource` instances.

```
public interface ResourceLoader {
    Resource getResource(String location);
}
```

All application contexts implement the `ResourceLoader` interface, and therefore all application contexts may be used to obtain `Resource` instances.

When you call `getResource()` on a specific application context, and the location path specified doesn't have a specific prefix, you will get back a `Resource` type that is appropriate to that particular application context. For example, assume the following snippet of code was executed against a `ClassPathXmlApplicationContext` instance:

```
Resource template = ctx.getResource("some/resource/path/myTemplate.txt);
```

What would be returned would be a `ClassPathResource`; if the same method was executed against a `FileSystemXmlApplicationContext` instance, you'd get back a `FileSystemResource`. For a `WebApplicationContext`, you'd get back a `ServletContextResource`, and so on.

As such, you can load resources in a fashion appropriate to the particular application context.

On the other hand, you may also force `ClassPathResource` to be used, regardless of the application context type, by specifying the special `classpath:` prefix:

```
Resource template = ctx.getResource("classpath:some/resource/path/myTemplate.txt);
```

Similarly, one can force a `UrlResource` to be used by specifying any of the standard `java.net.URL` prefixes:

```
Resource template = ctx.getResource("file:/some/resource/path/myTemplate.txt);
```

```
Resource template = ctx.getResource("http://myhost.com/resource/path/myTemplate.txt);
```

The following table summarizes the strategy for converting `String`s to `Resource`s:

Table 4.1. Resource strings

Prefix	Example	Explanation
classpath:	`classpath:com/myapp/config.xml`	Loaded from the classpath.
file:	`file:/data/config.xml`	Loaded as a `URL`, from the filesystem. [a]
http:	`http://myserver/logo.png`	Loaded as a `URL`.
(none)	`/data/config.xml`	Depends on the underlying `ApplicationContext`.

[a] But see also the section entitled Section 4.7.3, "`FileSystemResource` caveats".

4.5. The `ResourceLoaderAware` interface

The `ResourceLoaderAware` interface is a special marker interface, identifying objects that expect to be provided with a `ResourceLoader` reference.

```
public interface ResourceLoaderAware {

   void setResourceLoader(ResourceLoader resourceLoader);
}
```

When a class implements `ResourceLoaderAware` and is deployed into an application context (as a Spring-managed bean), it is recognized as `ResourceLoaderAware` by the application context. The application context will then invoke the `setResourceLoader(ResourceLoader)`, supplying itself as the argument (remember, all application contexts in Spring implement the `ResourceLoader` interface).

Of course, since an `ApplicationContext` is a `ResourceLoader`, the bean could also implement the `ApplicationContextAware` interface and use the supplied application context directly to load resources, but in general, it's better to use the specialized `ResourceLoader` interface if that's all that's needed. The code would just be coupled to the resource loading interface, which can be considered a utility interface, and not the whole Spring `ApplicationContext` interface.

4.6. `Resources` as dependencies

If the bean itself is going to determine and supply the resource path through some sort of dynamic process, it probably makes sense for the bean to use the `ResourceLoader` interface to load resources. Consider as an example the loading of a template of some sort, where the specific resource that is needed depends on the role of the user. If the resources are static, it makes sense to eliminate the use of the `ResourceLoader` interface completely, and just have the bean expose the `Resource` properties it needs, and expect that they will be injected into it.

What makes it trivial to then inject these properties, is that all application contexts register and use a special JavaBeans `PropertyEditor` which can convert `String` paths to `Resource` objects. So if `myBean` has a template property of type `Resource`, it can be configured with a simple string for that resource, as follows:

```
<bean id="myBean" class="...">
  <property name="template" value="some/resource/path/myTemplate.txt"/>
</bean>
```

Note that the resource path has no prefix, so because the application context itself is going to be used as the `ResourceLoader`, the resource itself will be loaded via a `ClassPathResource`, `FileSystemResource`, or `ServletContextResource` (as appropriate) depending on the exact type of the context.

If there is a need to force a specific `Resource` type to be used, then a prefix may be used. The following two examples show how to force a `ClassPathResource` and a `UrlResource` (the latter being used to access a filesystem file).

```
<property name="template" value="classpath:some/resource/path/myTemplate.txt">
```

```
<property name="template" value="file:/some/resource/path/myTemplate.txt"/>
```

4.7. Application contexts and `Resource` paths

4.7.1. Constructing application contexts

An application context constructor (for a specific application context type) generally takes a string or array of strings as the location path(s) of the resource(s) such as XML files that make up the definition of the context.

When such a location path doesn't have a prefix, the specific `Resource` type built from that path and used to load the bean definitions, depends on and is appropriate to the specific application context. For example, if you create a `ClassPathXmlApplicationContext` as follows:

```
ApplicationContext ctx = new ClassPathXmlApplicationContext("conf/appContext.xml");
```

The bean definitions will be loaded from the classpath, as a `ClassPathResource` will be used. But if you create

a `FileSystemXmlApplicationContext` as follows:

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/appContext.xml");
```

The bean definition will be loaded from a filesystem location, in this case relative to the current working directory.

Note that the use of the special classpath prefix or a standard URL prefix on the location path will override the default type of `Resource` created to load the definition. So this `FileSystemXmlApplicationContext`...

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("classpath:conf/appContext.xml");
```

... will actually load it's bean definitions from the classpath. However, it is still a `FileSystemXmlApplicationContext`. If it is subsequently used as a `ResourceLoader`, any unprefixed paths will still be treated as filesystem paths.

4.7.1.1. Constructing `ClassPathXmlApplicationContext` instances - shortcuts

The `ClassPathXmlApplicationContext` exposes a number of constructors to enable convenient instantiation. The basic idea is that one supplies merely a string array containing just the filenames of the XML files themselves (without the leading path information), and one *also* supplies a `Class`; the `ClassPathXmlApplicationContext` will derive the path information from the supplied class.

An example will hopefully make this clear. Consider a directory layout that looks like this:

```
com/
  foo/
    services.xml
    daos.xml
    MessengerService.class
```

A `ClassPathXmlApplicationContext` instance composed of the beans defined in the `'services.xml'` and `'daos.xml'` could be instantiated like so...

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] {"services.xml", "daos.xml"}, MessengerService.class);
```

Please do consult the Javadocs for the `ClassPathXmlApplicationContext` class for details of the various constructors.

4.7.2. Wildcards in application context constructor resource paths

The resource paths in application context constructor values may be a simple path (as shown above) which has a one-to-one mapping to a target Resource, or alternately may contain the special "classpath*:" prefix and/or internal Ant-style regular expressions (matched using Spring's `PathMatcher` utility). Both of the latter are effectively wildcards

One use for this mechanism is when doing component-style application assembly. All components can 'publish' context definition fragments to a well-known location path, and when the final application context is created using the same path prefixed via `classpath*:`, all component fragments will be picked up automatically.

Note that this wildcarding is specific to use of resource paths in application context constructors (or when using the `PathMatcher` utility class hierarchy directly), and is resolved at construction time. It has nothing to do with

the `Resource` type itself. It's not possible to use the `classpath*:` prefix to construct an actual `Resource`, as a resource points to just one resource at a time.

4.7.2.1. Ant-style Patterns

When the path location contains an Ant-style pattern, for example:

```
/WEB-INF/*-context.xml
com/mycompany/**/applicationContext.xml
file:C:/some/path/*-context.xml
classpath:com/mycompany/**/applicationContext.xml
```

... the resolver follows a more complex but defined procedure to try to resolve the wildcard. It produces a Resource for the path up to the last non-wildcard segment and obtains a URL from it. If this URL is not a "jar:" URL or container-specific variant (e.g. "`zip:`" in WebLogic, "`wsjar`" in WebSphere, etc.), then a `java.io.File` is obtained from it, and used to resolve the wildcard by walking the filesystem. In the case of a jar URL, the resolver either gets a `java.net.JarURLConnection` from it, or manually parse the jar URL, and then traverse the contents of the jar file, to resolve the wildcards.

4.7.2.1.1. Implications on portability

If the specified path is already a file URL (either explicitly, or implicitly because the base `ResourceLoader` is a filesystem one, then wildcarding is guaranteed to work in a completely portable fashion.

If the specified path is a classpath location, then the resolver must obtain the last non-wildcard path segment URL via a `Classloader.getResource()` call. Since this is just a node of the path (not the file at the end) it is actually undefined (in the `ClassLoader` Javadocs) exactly what sort of a URL is returned in this case. In practice, it is always a `java.io.File` representing the directory, where the classpath resource resolves to a filesystem location, or a jar URL of some sort, where the classpath resource resolves to a jar location. Still, there is a portability concern on this operation.

If a jar URL is obtained for the last non-wildcard segment, the resolver must be able to get a `java.net.JarURLConnection` from it, or manually parse the jar URL, to be able to walk the contents of the jar, and resolve the wildcard. This will work in most environments, but will fail in others, and it is strongly recommended that the wildcard resolution of resources coming from jars be thoroughly tested in your specific environment before you rely on it.

4.7.2.2. The `classpath*:` prefix

When constructing an XML-based application context, a location string may use the special `classpath*:` prefix:

```
ApplicationContext ctx =
    new ClassPathXmlApplicationContext("classpath*:conf/appContext.xml");
```

This special prefix specifies that all classpath resources that match the given name must be obtained (internally, this essentially happens via a `ClassLoader.getResources(...)` call), and then merged to form the final application context definition.

Classpath*: portability

The wildcard classpath relies on the `getResources()` method of the underlying classloader. As most application servers nowadays supply their own classloader implementation, the behavior might differ especially when dealing with jar files. A simple test to check if `classpath*` works is to

use the classloader to load a file from within a jar on the classpath: `getClass().getClassLoader().getResources("<someFileInsideTheJar>")`. Try this test with files that have the same name but are placed inside two different locations. In case an inappropriate result is returned, check the application server documentation for settings that might affect the classloader behavior.

The "`classpath*:`" prefix can also be combined with a `PathMatcher` pattern in the rest of the location path, for example "`classpath*:META-INF/*-beans.xml`". In this case, the resolution strategy is fairly simple: a ClassLoader.getResources() call is used on the last non-wildcard path segment to get all the matching resources in the class loader hierarchy, and then off each resource the same PathMatcher resoltion strategy described above is used for the wildcard subpath.

4.7.2.3. Other notes relating to wildcards

Please note that "`classpath*:`" when combined with Ant-style patterns will only work reliably with at least one root directory before the pattern starts, unless the actual target files reside in the file system. This means that a pattern like "`classpath*:*.xml`" will not retrieve files from the root of jar files but rather only from the root of expanded directories. This originates from a limitation in the JDK's `ClassLoader.getResources()` method which only returns file system locations for a passed-in empty string (indicating potential roots to search).

Ant-style patterns with "`classpath:`" resources are not guaranteed to find matching resources if the root package to search is available in multiple class path locations. This is because a resource such as

```
    com/mycompany/package1/service-context.xml
```

may be in only one location, but when a path such as

```
    classpath:com/mycompany/**/service-context.xml
```

is used to try to resolve it, the resolver will work off the (first) URL returned by `getResource("com/mycompany");`. If this base package node exists in multiple classloader locations, the actual end resource may not be underneath. Therefore, preferably, use "`classpath*:`" with the same Ant-style pattern in such a case, which will search all class path locations that contain the root package.

4.7.3. `FileSystemResource` caveats

A `FileSystemResource` that is not attached to a `FileSystemApplicationContext` (that is, a `FileSystemApplicationContext` is not the actual `ResourceLoader`) will treat absolute vs. relative paths as you would expect. Relative paths are relative to the current working directory, while absolute paths are relative to the root of the filesystem.

For backwards compatibility (historical) reasons however, this changes when the `FileSystemApplicationContext` is the `ResourceLoader`. The `FileSystemApplicationContext` simply forces all attached `FileSystemResource` instances to treat all location paths as relative, whether they start with a leading slash or not. In practice, this means the following are equivalent:

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("conf/context.xml");
```

```
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("/conf/context.xml");
```

As are the following: (Even though it would make sense for them to be different, as one case is relative and the other absolute.)

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("some/resource/path/myTemplate.txt");
```

```
FileSystemXmlApplicationContext ctx = ...;
ctx.getResource("/some/resource/path/myTemplate.txt");
```

In practice, if true absolute filesystem paths are needed, it is better to forgo the use of absolute paths with `FileSystemResource` / `FileSystemXmlApplicationContext`, and just force the use of a `UrlResource`, by using the `file:` URL prefix.

```
// actual context type doesn't matter, the Resource will always be UrlResource
ctx.getResource("file:/some/resource/path/myTemplate.txt");
```

```
// force this FileSystemXmlApplicationContext to load it's definition via a UrlResource
ApplicationContext ctx =
    new FileSystemXmlApplicationContext("file:/conf/context.xml");
```

Chapter 5. Validation, Data-binding, the `BeanWrapper`, and `PropertyEditors`

5.1. Introduction

There are pros and cons for considering validation as business logic, and Spring offers a design for validation (and data binding) that does not exclude either one of them. Specifically validation should not be tied to the web tier, should be easy to localize and it should be possible to plug in any validator available. Considering the above, Spring has come up with a `Validator` interface that is both basic and eminently 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 in order. We're explaining the `BeanWrapper` in this chapter since if you were going to use it at all, you would probably do so 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`.

5.2. Validation using Spring's `Validator` interface

Spring's features a `Validator` interface that you can use to validate objects. The `Validator` interface works using an `Errors` object so that while validating, validators can report validation failures to the `Errors` object.

Let's consider a small data object:

```
public class Person {

  private String name;
  private int age;

  // the usual getters and setters...
}
```

We're going to provide validation behavior for the `Person` class by implementing the following two methods of the `org.springframework.validation.Validator` interface:

- `supports(Class)` - Can this `Validator` validate instances of the supplied `Class`?
- `validate(Object, org.springframework.validation.Errors)` - validates the given object and in case of validation errors, registers those with the given `Errors` object

Implementing a `Validator` is fairly straightforward, especially when you know of the `ValidationUtils` helper class that the Spring Framework also provides.

```
public class PersonValidator implements Validator {

    /**
    * This Validator validates just Person instances
    */
    public boolean supports(Class clazz) {
        return Person.class.equals(clazz);
    }

    public void validate(Object obj, Errors e) {
        ValidationUtils.rejectIfEmpty(e, "name", "name.empty");
        Person p = (Person) obj;
        if (p.getAge() < 0) {
            e.rejectValue("age", "negativevalue");
        } else if (p.getAge() > 110) {
            e.rejectValue("age", "too.darn.old");
        }
    }
}
```

As you can see, the `static rejectIfEmpty(..)` method on the `ValidationUtils` class is used to reject the `'name'` property if it is `null` or the empty string. Have a look at the Javadoc for the `ValidationUtils` class to see what functionality it provides besides the example shown previously.

While it is certainly possible to implement a single `Validator` class to validate each of the nested objects in a rich object, it may be better to encapsulate the validation logic for each nested class of object in its own `Validator` implementation. A simple example of a *'rich'* object would be a `Customer` that is composed of two `String` properties (a first and second name) and a complex `Address` object. `Address` objects may be used independently of `Customer` objects, and so a distinct `AddressValidator` has been implemented. If you want your `CustomerValidator` to reuse the logic contained within the `AddressValidator` class without recourse to copy-n-paste you can dependency-inject or instantiate an `AddressValidator` within your `CustomerValidator`, and use it like so:

```
public class CustomerValidator implements Validator {

   private final Validator addressValidator;

   public CustomerValidator(Validator addressValidator) {
      if (addressValidator == null) {
          throw new IllegalArgumentException("The supplied [Validator] is required and must not be null.");
      }
      if (!addressValidator.supports(Address.class)) {
          throw new IllegalArgumentException(
            "The supplied [Validator] must support the validation of [Address] instances.");
      }
      this.addressValidator = addressValidator;
   }

    /**
    * This Validator validates Customer instances, and any subclasses of Customer too
    */
   public boolean supports(Class clazz) {
      return Customer.class.isAssignableFrom(clazz);
   }

   public void validate(Object target, Errors errors) {
      ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "field.required");
      ValidationUtils.rejectIfEmptyOrWhitespace(errors, "surname", "field.required");
      Customer customer = (Customer) target;
      try {
          errors.pushNestedPath("address");
          ValidationUtils.invokeValidator(this.addressValidator, customer.getAddress(), errors);
      } finally {
          errors.popNestedPath();
      }
   }
}
```

Validation errors are reported to the `Errors` object passed to the validator. In case of Spring Web MVC you can use `<spring:bind/>` tag to inspect the error messages, but of course you can also inspect the errors object yourself. More information about the methods it offers can be found from the Javadoc.

5.3. Resolving codes to error messages

We've talked about databinding and validation. Outputting messages corresponding to validation errors is the last thing we need to discuss. In the example we've shown above, we rejected the `name` and the `age` field. If we're going to output the error messages by using a `MessageSource`, we will do so using the error code we've given when rejecting the field ('name' and 'age' in this case). When you call (either directly, or indirectly, using for example the `ValidationUtils` class) `rejectValue` or one of the other `reject` methods from the `Errors` interface, the underlying implementation will not only register the code you've passed in, but also a number of additional error codes. What error codes it registers is determined by the `MessageCodesResolver` that is used. By default, the `DefaultMessageCodesResolver` is used, which for example not only registers a message with the code you gave, but also messages that include the field name you passed to the reject method. So in case you reject a field using `rejectValue("age", "too.darn.old")`, apart from the `too.darn.old` code, Spring will also register `too.darn.old.age` and `too.darn.old.age.int` (so the first will include the field name and the second will include the type of the field); this is done as a convenience to aid developers in targeting error messages and suchlike.

More information on the `MessageCodesResolver` and the default strategy can be found online with the Javadocs for MessageCodesResolver and DefaultMessageCodesResolver respectively.

5.4. 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 `bingoMadness` has a setter `setBingoMadness(..)` and a getter `getBingoMadness()`. For more information about JavaBeans and the specification, please refer to Sun's website (java.sun.com/products/javabeans).

One quite important concept of the beans package is the `BeanWrapper` interface and its corresponding implementation (`BeanWrapperImpl`). As quoted from the Javadoc, the `BeanWrapper` offers functionality to set and get property values (individually or in bulk), get property descriptors, and to query properties to determine if they are readable or writable. Also, the `BeanWrapper` offers support for nested properties, enabling the setting of properties on sub-properties to an unlimited depth. Then, the `BeanWrapper` supports the ability to add standard JavaBeans `PropertyChangeListeners` and `VetoableChangeListeners`, without the need for supporting code in the target class. Last but not least, the `BeanWrapper` provides support for the setting of indexed properties. The `BeanWrapper` usually isn't used by application code directly, but by the `DataBinder` and the `BeanFactory`.

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

5.4.1. 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 5.1. 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
`account[COMPANYNAME]`	Indicates the value of the map entry indexed by the key *COMPANYNAME* of the Map property `account`

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

(This next section is not vitally 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, you should skip ahead to the section 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 String name;
    private float salary;

    public String getName() {
        return this.name;
    }
    public void setName(String name) {
        this.name = name;
    }
    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`:

```
BeanWrapper company = BeanWrapperImpl(new Company());
```

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

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

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

5.4.2. Built-in PropertyEditor implementations

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` or alternately in a specific IoC container as mentioned in the previous chapter, 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. Most, but not all (as indicated below), are registered by default by `BeanWrapperImpl`. Where the property editor is configurable in some fashion, you can of course still register your own variant to override the default one:

Table 5.2. Built-in `PropertyEditors`

Class	Explanation
`ByteArrayPropertyEditor`	Editor for byte arrays. Strings will simply be converted to their corresponding byte representations. Registered by default by `BeanWrapperImpl`.
`ClassEditor`	Parses Strings representing classes to actual classes and the other way around. When a class is not found, an `IllegalArgumentException` is thrown. Registered by default by `BeanWrapperImpl`.
`CustomBooleanEditor`	Customizable property editor for `Boolean` properties. Registered by default by `BeanWrapperImpl`, but, can be overridden by registering custom instance of it as custom editor.
`CustomCollectionEditor`	Property editor for Collections, converting any source `Collection` to a given target `Collection` type.

Class	Explanation
CustomDateEditor	Customizable property editor for java.util.Date, supporting a custom DateFormat. NOT registered by default. Must be user registered as needed with appropriate format.
CustomNumberEditor	Customizable property editor for any Number subclass like Integer, Long, Float, Double. Registered by default by BeanWrapperImpl, but can be overridden by registering custom instance of it as a custom editor.
FileEditor	Capable of resolving Strings to java.io.File objects. Registered by default by BeanWrapperImpl.
InputStreamEditor	One-way property editor, capable of taking a text string and producing (via an intermediate ResourceEditor and Resource) an InputStream, so InputStream properties may be directly set as Strings. Note that the default usage will not close the InputStream for you! Registered by default by BeanWrapperImpl.
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). Registered by default by BeanWrapperImpl.
PatternEditor	Capable of resolving Strings to JDK 1.5 Pattern objects and vice versa.
PropertiesEditor	Capable of converting Strings (formatted using the format as defined in the Javadoc for the java.lang.Properties class) to Properties objects. Registered by default by BeanWrapperImpl.
StringTrimmerEditor	Property editor that trims Strings. Optionally allows transforming an empty string into a null value. NOT registered by default; must be user registered as needed.
URLEditor	Capable of resolving a String representation of a URL to an actual URL object. Registered by default by BeanWrapperImpl.

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 PropertyEditor implementations for types such as Font, Color, and most of the primitive types. Note also that the standard JavaBeans infrastructure will automatically discover PropertyEditor classes (without you having to register them explicitly) if they are in the same package as the class they handle, and have the same name as that class, with 'Editor' appended; for example, one could have the following class and package structure, which would be sufficient for the FooEditor class to be recognized and used as the PropertyEditor for Foo-typed properties.

```
com
  chank
    pop
      Foo
      FooEditor   // the PropertyEditor for the Foo class
```

Note that you can also use the standard BeanInfo JavaBeans mechanism here as well (described in not-amazing-detail here). Find below an example of using the BeanInfo mechanism for explicitly registering one or more PropertyEditor instances with the properties of an associated class.

```
com
  chank
    pop
      Foo
      FooBeanInfo   // the BeanInfo for the Foo class
```

Here is the Java source code for the referenced `FooBeanInfo` class. This would associate a `CustomNumberEditor` with the `age` property of the `Foo` class.

```
public class FooBeanInfo extends SimpleBeanInfo {

    public PropertyDescriptor[] getPropertyDescriptors() {
        try {
            final PropertyEditor numberPE = new CustomNumberEditor(Integer.class, true);
            PropertyDescriptor ageDescriptor = new PropertyDescriptor("age", Foo.class) {
                public PropertyEditor createPropertyEditor(Object bean) {
                    return numberPE;
                };
            };
            return new PropertyDescriptor[] { ageDescriptor };
        }
        catch (IntrospectionException ex) {
            throw new Error(ex.toString());
        }
    }
}
```

5.4.2.1. Registering additional custom `PropertyEditors`

When setting bean properties as a string value, a Spring IoC container ultimately uses standard JavaBeans `PropertyEditors` to convert these Strings to the complex type of the property. Spring pre-registers a number of custom `PropertyEditors` (for example, to convert a classname expressed as a string into a real `Class` object). Additionally, Java's standard JavaBeans `PropertyEditor` lookup mechanism allows a `PropertyEditor` for a class simply to be named appropriately and placed in the same package as the class it provides support for, to be found automatically.

If there is a need to register other custom `PropertyEditors`, there are several mechanisms available. The most manual approach, which is not normally convenient or recommended, is to simply use the `registerCustomEditor()` method of the `ConfigurableBeanFactory` interface, assuming you have a `BeanFactory` reference. The more convenient mechanism is to use a special bean factory post-processor called `CustomEditorConfigurer`. Although bean factory post-processors can be used semi-manually with `BeanFactory` implementations, this one has a nested property setup, so it is strongly recommended that it is used with the `ApplicationContext`, where it may be deployed in similar fashion to any other bean, and automatically detected and applied.

Note that all bean factories and application contexts automatically use a number of built-in property editors, through their use of something called a `BeanWrapper` to handle property conversions. The standard property editors that the `BeanWrapper` registers are listed in the previous section. Additionally, `ApplicationContexts` also override or add an additional number of editors to handle resource lookups in a manner appropriate to the specific application context type.

Standard JavaBeans `PropertyEditor` instances are used to convert property values expressed as strings to the actual complex type of the property. `CustomEditorConfigurer`, a bean factory post-processor, may be used to conveniently add support for additional `PropertyEditor` instances to an `ApplicationContext`.

Consider a user class `ExoticType`, and another class `DependsOnExoticType` which needs `ExoticType` set as a property:

```
package example;
```

```
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;
    }
}
```

When things are properly set up, we want to be able to assign the type property as a string, which a PropertyEditor will behind the scenes convert into a real ExoticType object:

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

The PropertyEditor implementation could look similar to this:

```
// converts string representation to ExoticType object
package example;

public class ExoticTypeEditor extends PropertyEditorSupport {

    private String format;

    public void setFormat(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);
    }
}
```

Finally, we use CustomEditorConfigurer to register the new PropertyEditor with the ApplicationContext, which will then be able to use it as needed:

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

5.4.2.1.1. Using PropertyEditorRegistrars

Another mechanism for registering property editors with the Spring container is to create and use a PropertyEditorRegistrar. This interface is particularly useful when you need to use the same set of property

editors in several different situations: write a corresponding registrar and reuse that in each case. PropertyEditorRegistrars work in conjunction with an interface called PropertyEditorRegistry, an interface that is implemented by the Spring BeanWrapper (and DataBinder). PropertyEditorRegistrars are particularly convenient when used in conjunction with the CustomEditorConfigurer (introduced here), which exposes a property called setPropertyEditorRegistrars(..): PropertyEditorRegistrars added to a CustomEditorConfigurer in this fashion can easily be shared with DataBinder and Spring MVC Controllers. Furthermore, it avoids the need for synchronization on custom editors: a PropertyEditorRegistrar is expected to create fresh PropertyEditor instances for each bean creation attempt.

Using a PropertyEditorRegistrar is perhaps best illustrated with an example. First off, you need to create your own PropertyEditorRegistrar implementation:

```
package com.foo.editors.spring;

public final class CustomPropertyEditorRegistrar implements PropertyEditorRegistrar {

    public void registerCustomEditors(PropertyEditorRegistry registry) {

        // it is expected that new PropertyEditor instances are created
        registry.registerCustomEditor(ExoticType.class, new ExoticTypeEditor());

        // you could register as many custom property editors as are required here...
    }
}
```

See also the org.springframework.beans.support.ResourceEditorRegistrar for an example PropertyEditorRegistrar implementation. Notice how in its implementation of the registerCustomEditors(..) method it creates new instances of each property editor.

Next we configure a CustomEditorConfigurer and inject an instance of our CustomPropertyEditorRegistrar into it:

```
<bean class="org.springframework.beans.factory.config.CustomEditorConfigurer">
    <property name="propertyEditorRegistrars">
        <list>
            <ref bean="customPropertyEditorRegistrar"/>
        </list>
    </property>
</bean>

<bean id="customPropertyEditorRegistrar" class="com.foo.editors.spring.CustomPropertyEditorRegistrar"/>
```

Finally, and in a bit of a departure from the focus of this chapter, for those of you using Spring's MVC web framework, using PropertyEditorRegistrars in conjunction with data-binding Controllers (such as SimpleFormController) can be very convenient. Find below an example of using a PropertyEditorRegistrar in the implementation of an initBinder(..) method:

```
public final class RegisterUserController extends SimpleFormController {

    private final PropertyEditorRegistrar customPropertyEditorRegistrar;

    public RegisterUserController(PropertyEditorRegistrar propertyEditorRegistrar) {
        this.customPropertyEditorRegistrar = propertyEditorRegistrar;
    }

    protected void initBinder(HttpServletRequest request, ServletRequestDataBinder binder) throws Exception {
        this.customPropertyEditorRegistrar.registerCustomEditors(binder);
    }

    // other methods to do with registering a User
}
```

This style of PropertyEditor registration can lead to concise code (the implementation of initBinder(..) is

just one line long!), and allows common PropertyEditor registration code to be encapsulated in a class and then shared amongst as many Controllers as needed.

Chapter 6. Aspect Oriented Programming with Spring

6.1. Introduction

Aspect-Oriented Programming (*AOP*) complements Object-Oriented Programming (OOP) by providing another way of thinking about program structure. In addition to classes, AOP gives you *aspects*. Aspects enable modularization of concerns such as transaction management that cut across multiple types and objects. (Such concerns are often termed *crosscutting* concerns.)

One of the key components of Spring is the *AOP framework*. While the Spring IoC container does 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.

> **Spring 2.0 AOP**
>
> Spring 2.0 introduces a simpler and more powerful way of writing custom aspects using either a schema-based approach or the @AspectJ annotation style. Both of these styles offer fully typed advice and use of the AspectJ pointcut language, while still using Spring AOP for weaving.
>
> The Spring 2.0 schema and @AspectJ based AOP support is discussed in this chapter. Spring 2.0 AOP remains fully backwards compatible with Spring 1.2 AOP, and the lower-level AOP support offered by the Spring 1.2 APIs is discussed in the following chapter.

AOP is used in the Spring Framework:

- 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 the Spring Framework's transaction abstraction.
- To allow users to implement custom aspects, complementing their use of OOP with AOP.

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.

6.1.1. 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 that cuts across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. In Spring AOP, aspects are implemented using regular classes (the schema-based approach) or regular classes annotated with the `@Aspect` annotation (`@AspectJ` style).
- *Join point*: A point during the execution of a program, such as the execution of a method or the handling of an exception. In Spring AOP, a join point *always* represents a method execution. Join point information is

available in advice bodies by declaring a parameter of type `org.aspectj.lang.JoinPoint`.

- *Advice*: Action taken by an aspect at a particular join point. Different types of advice include "around," "before" and "after" advice. Advice types are discussed below. Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the join point.

- *Pointcut*: A predicate that matches join points. Advice is associated with a pointcut expression and runs at any join point matched by the pointcut (for example, the execution of a method with a certain name). The concept of join points as matched by pointcut expressions is central to AOP: Spring uses the AspectJ pointcut language by default.

- *Introduction*: (Also known as an inter-type declaration). Declaring additional methods or fields on behalf of a type. Spring AOP allows you to introduce new interfaces (and a corresponding implementation) to any proxied object. For example, you could use an introduction to make a bean implement an `IsModified` interface, to simplify caching.

- *Target object*: Object being advised by one or more aspects. Also referred to as the *advised* object. Since Spring AOP is implemented using runtime proxies, this object will always be a *proxied* object.

- *AOP proxy*: An object created by the AOP framework in order to implement the aspect contracts (advise method executions and so on). In the Spring Framework, an AOP proxy will be a JDK dynamic proxy or a CGLIB proxy. *Proxy creation is transparent to users of the schema-based and @AspectJ styles of aspect declaration introduced in Spring 2.0.*

- *Weaving*: Linking aspects with other application types or objects to create an advised object. This can be done at compile time (using the AspectJ compiler, for example), load time, or at runtime. Spring AOP, like other pure Java AOP frameworks, performs weaving at runtime.

Types of advice:

- *Before advice*: Advice that executes before a join point, but which does not have the ability to prevent execution flow proceeding to the join point (unless it throws an exception).

- *After returning advice*: Advice to be executed after a join point completes normally: for example, if a method returns without throwing an exception.

- *After throwing advice*: Advice to be executed if a method exits by throwing an exception.

- *After (finally) advice*: Advice to be executed regardless of the means by which a join point exits (normal or exceptional return).

- *Around advice*: Advice that surrounds a join point such as a method invocation. This is the most powerful kind of advice. Around advice can perform custom behavior before and after the method invocation. It is also responsible for choosing whether to proceed to the join point or to shortcut the advised method execution by returning its own return value or throwing an exception.

Around advice is the most general kind of advice. Since Spring AOP, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required behavior. 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 do not need to invoke the `proceed()` method on the `JoinPoint` used for around advice, and hence cannot fail to invoke it.

In Spring 2.0, all advice parameters are statically typed, so that you work with advice parameters of the appropriate type (the type of the return value from a method execution for example) rather than `Object` arrays.

The concept of join points, matched by pointcuts, is the key to AOP which distinguishes it from older technologies offering only interception. Pointcuts enable advice to be targeted independently of the Object-Oriented hierarchy. For example, an around advice providing declarative transaction management can be applied to a set of methods spanning multiple objects (such as all business operations in the service layer).

6.1.2. Spring AOP capabilities and goals

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 AOP currently supports only method execution join points (advising the execution of methods on Spring beans). Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs. If you need to advise field access and update join points, consider a language such as AspectJ.

Spring AOP'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, the Spring Framework's AOP functionality is normally used in conjunction with the Spring IoC container. Aspects are configured using normal bean definition syntax (although this allows powerful "autoproxying" capabilities): this is a crucial difference from other AOP implementations. There are some things you cannot do easily or efficiently with Spring AOP, such as advise very fine-grained objects: AspectJ is 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.

Spring AOP will never strive to compete with AspectJ to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring AOP and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Spring 2.0 seamlessly integrates Spring AOP and IoC with AspectJ, to enable all uses of AOP to be catered for within a consistent Spring-based application architecture. This integration does not affect the Spring AOP API or the AOP Alliance API: Spring AOP remains backward-compatible. See the following chapter for a discussion of the Spring AOP APIs.

Note

One of the central tenets of the Spring Framework is that of *non-invasiveness*; this is the idea that you should not be forced to introduce framework-specific classes and interfaces into your business/domain model. However, in some places the Spring Framework does give you the option to introduce Spring Framework-specific dependencies into your codebase: the rationale in giving you such options is because in certain scenarios it might be just plain easier to read or code some specific piece of functionality in such a way. The Spring Framework (almost) always offers you the choice though: you have the freedom to make an informed decision as to which option best suits your particular use case or scenario.

One such choice that is relevant to this chapter is that of which AOP framework (and which AOP style) to choose. You have the choice of AspectJ and/or Spring AOP, and you also have the choice of either the @AspectJ annotation-style approach or the Spring XML configuration-style approach. The fact that this chapter chooses to introduce the @AspectJ-style approach first should not be

taken as an indication that the Spring team favors the @AspectJ annotation-style approach over the Spring XML configuration-style.

See the section entitled Section 6.4, “Choosing which AOP declaration style to use” for a fuller discussion of the whys and wherefores of each style.

6.1.3. AOP Proxies

Spring AOP defaults to using standard J2SE *dynamic proxies* for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes, rather than interfaces. CGLIB is used by default if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes, business classes normally will implement one or more business interfaces. It is possible to force the use of CGLIB [133], in those (hopefully rare) cases where you need to advise a method that is not declared on an interface, or where you need to pass a proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is *proxy-based*. See the section entitled Section 6.6.1, “Understanding AOP proxies” for a thorough examination of exactly what this implementation detail actually means.

6.2. @AspectJ support

@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations. The @AspectJ style was introduced by the AspectJ project as part of the AspectJ 5 release. Spring 2.0 interprets the same annotations as AspectJ 5, using a library supplied by AspectJ for pointcut parsing and matching. The AOP runtime is still pure Spring AOP though, and there is no dependency on the AspectJ compiler or weaver.

Using the AspectJ compiler and weaver enables use of the full AspectJ language, and is discussed in Section 6.8, “Using AspectJ with Spring applications”.

6.2.1. Enabling @AspectJ Support

To use @AspectJ aspects in a Spring configuration you need to enable Spring support for configuring Spring AOP based on @AspectJ aspects, and *autoproxying* beans based on whether or not they are advised by those aspects. By autoproxying we mean that if Spring determines that a bean is advised by one or more aspects, it will automatically generate a proxy for that bean to intercept method invocations and ensure that advice is executed as needed.

The @AspectJ support is enabled by including the following element inside your spring configuration:

```
<aop:aspectj-autoproxy/>
```

This assumes that you are using schema support as described in Appendix A, *XML Schema-based configuration*. See Section A.2.6, “The `aop` schema” for how to import the tags in the aop namespace.

If you are using the DTD, it is still possible to enable @AspectJ support by adding the following definition to your application context:

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator" />
```

You will also need two AspectJ libraries on the classpath of your application: `aspectjweaver.jar` and `aspectjrt.jar`. These libraries are available in the `'lib'` directory of an AspectJ installation (version 1.5.1 or later required), or in the `'lib/aspectj'` directory of the Spring-with-dependencies distribution.

6.2.2. Declaring an aspect

With the @AspectJ support enabled, any bean defined in your application context with a class that is an @AspectJ aspect (has the `@Aspect` annotation) will be automatically detected by Spring and used to configure Spring AOP. The following example shows the minimal definition required for a not-very-useful aspect:

A regular bean definition in the application context, pointing to a bean class that has the `@Aspect` annotation:

```
<bean id="myAspect" class="org.xyz.NotVeryUsefulAspect">
   <!-- configure properties of aspect here as normal -->
</bean>
```

And the `NotVeryUsefulAspect` class definition, annotated with `org.aspectj.lang.annotation.Aspect` annotation;

```
package org.xyz;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class NotVeryUsefulAspect {

}
```

Aspects (classes annotated with `@Aspect`) may have methods and fields just like any other class. They may also contain pointcut, advice, and introduction (inter-type) declarations.

Advising aspects

In Spring AOP, it is *not* possible to have aspects themselves be the target of advice from other aspects. The *@Aspect* annotation on a class marks it as an aspect, and hence excludes it from auto-proxying.

6.2.3. Declaring a pointcut

Recall that pointcuts determine join points of interest, and thus enable us to control when advice executes. *Spring AOP only supports method execution join points for Spring beans*, so you can think of a pointcut as matching the execution of methods on Spring beans. A pointcut declaration has two parts: a signature comprising a name and any parameters, and a pointcut expression that determines *exactly* which method executions we are interested in. In the @AspectJ annotation-style of AOP, a pointcut signature is provided by a regular method definition, and the pointcut expression is indicated using the `@Pointcut` annotation (the method serving as the pointcut signature *must* have a `void` return type).

An example will help make this distinction between a pointcut signature and a pointcut expression clear. The following example defines a pointcut named `'anyOldTransfer'` that will match the execution of any method named `'transfer'`:

```
@Pointcut("execution(* transfer(..))")// the pointcut expression
private void anyOldTransfer() {}// the pointcut signature
```

The pointcut expression that forms the value of the `@Pointcut` annotation is a regular AspectJ 5 pointcut expression. For a full discussion of AspectJ's pointcut language, see the AspectJ Programming Guide (and for Java 5 based extensions, the AspectJ 5 Developers Notebook) or one of the books on AspectJ such as “Eclipse AspectJ” by Colyer et. al. or “AspectJ in Action” by Ramnivas Laddad.

6.2.3.1. Supported Pointcut Designators

Spring AOP supports the following AspectJ pointcut designators for use in pointcut expressions:

Other pointcut types

The full AspectJ pointcut language supports additional pointcut designators that are not supported in Spring. These are: `call`, `initialization`, `preinitialization`, `staticinitialization`, `get`, `set`, `handler`, `adviceexecution`, `withincode`, `cflow`, `cflowbelow`, `if`, `@this`, and `@withincode`. Use of these pointcut designators in pointcut expressions interpreted by Spring AOP will result in an `IllegalArgumentException` being thrown.

The set of pointcut designators supported by Spring AOP may be extended in future releases both to support more of the AspectJ pointcut designators (e.g. "if"), and potentially to support Spring specific designators such as "bean" (matching on bean name).

- *execution* - for matching method execution join points, this is the primary pointcut designator you will use when working with Spring AOP
- *within* - limits matching to join points within certain types (simply the execution of a method declared within a matching type when using Spring AOP)
- *this* - limits matching to join points (the execution of methods when using Spring AOP) where the bean reference (Spring AOP proxy) is an instance of the given type
- *target* - limits matching to join points (the execution of methods when using Spring AOP) where the target object (application object being proxied) is an instance of the given type
- *args* - limits matching to join points (the execution of methods when using Spring AOP) where the arguments are instances of the given types
- `@target` - limits matching to join points (the execution of methods when using Spring AOP) where the class of the executing object has an annotation of the given type
- `@args` - limits matching to join points (the execution of methods when using Spring AOP) where the runtime type of the actual arguments passed have annotations of the given type(s)
- `@within` - limits matching to join points within types that have the given annotation (the execution of methods declared in types with the given annotation when using Spring AOP)
- *@annotation* - limits matching to join points where the subject of the join point (method being executed in Spring AOP) has the given annotation

Because Spring AOP limits matching to only method execution join points, the discussion of the pointcut designators above gives a narrower definition than you will find in the AspectJ programming guide. In addition, AspectJ itself has type-based semantics and at an execution join point both 'this' and 'target' refer to the same object - the object executing the method. Spring AOP is a proxy based system and differentiates between the proxy object itself (bound to 'this') and the target object behind the proxy (bound to 'target').

6.2.3.2. Combining pointcut expressions

Pointcut expressions can be combined using '&&', '||' and '!'. It is also possible to refer to pointcut expressions by name. The following example shows three pointcut expressions: `anyPublicOperation` (which matches if a method execution join point represents the execution of any public method); `inTrading` (which matches if a method execution is in the trading module), and `tradingOperation` (which matches if a method execution represents any public method in the trading module).

```
    @Pointcut("execution(public * *(..))")
    private void anyPublicOperation() {}

    @Pointcut("within(com.xyz.someapp.trading..*")
    private void inTrading() {}

    @Pointcut("anyPublicOperation() && inTrading()")
    private void tradingOperation() {}
```

It is a best practice to build more complex pointcut expressions out of smaller named components as shown above. When referring to pointcuts by name, normal Java visibility rules apply (you can see private pointcuts in the same type, protected pointcuts in the hierarchy, public pointcuts anywhere and so on). Visibility does not affect pointcut *matching*.

6.2.3.3. Sharing common pointcut definitions

When working with enterprise applications, you often want to refer to modules of the application and particular sets of operations from within several aspects. We recommend defining a "SystemArchitecture" aspect that captures common pointcut expressions for this purpose. A typical such aspect would look as follows:

```
package com.xyz.someapp;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class SystemArchitecture {

  /**
   * A join point is in the web layer if the method is defined
   * in a type in the com.xyz.someapp.web package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.web..*)")
  public void inWebLayer() {}

  /**
   * A join point is in the service layer if the method is defined
   * in a type in the com.xyz.someapp.service package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.service..*)")
  public void inServiceLayer() {}

  /**
   * A join point is in the data access layer if the method is defined
   * in a type in the com.xyz.someapp.dao package or any sub-package
   * under that.
   */
  @Pointcut("within(com.xyz.someapp.dao..*)")
  public void inDataAccessLayer() {}

  /**
   * A business service is the execution of any method defined on a service
   * interface. This definition assumes that interfaces are placed in the
   * "service" package, and that implementation types are in sub-packages.
   *
   * If you group service interfaces by functional area (for example,
   * in packages com.xyz.someapp.abc.service and com.xyz.def.service) then
   * the pointcut expression "execution(* com.xyz.someapp..service.*.*(..))"
```

```
    * could be used instead.
    */
  @Pointcut("execution(* com.xyz.someapp.service.*.*(..))")
  public void businessService() {}

  /**
    * A data access operation is the execution of any method defined on a
    * dao interface. This definition assumes that interfaces are placed in the
    * "dao" package, and that implementation types are in sub-packages.
    */
  @Pointcut("execution(* com.xyz.someapp.dao.*.*(..))")
  public void dataAccessOperation() {}

}
```

The pointcuts defined in such an aspect can be referred to anywhere that you need a pointcut expression. For example, to make the service layer transactional, you could write:

```
<aop:config>
  <aop:advisor
      pointcut="com.xyz.someapp.SystemArchitecture.businessService()"
      advice-ref="tx-advice"/>
</aop:config>

<tx:advice id="tx-advice">
<tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

The `<aop:config>` and `<aop:advisor>` tags are discussed in the section entitled Section 6.3, “Schema-based AOP support”. The transaction tags are discussed in the chapter entitled Chapter 9, *Transaction management*.

6.2.3.4. Examples

Spring AOP users are likely to use the `execution` pointcut designator the most often. The format of an execution expression is:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern)
          throws-pattern?)
```

All parts except the returning type pattern (ret-type-pattern in the snippet above), name pattern, and parameters pattern are optional. The returning type pattern determines what the return type of the method must be in order for a join point to be matched. Most frequently you will use `*` as the returning type pattern, which matches any return type. A fully-qualified type name will match only when the method returns the given type. The name pattern matches the method name. You can use the `*` wildcard as all or part of a name pattern. The parameters pattern is slightly more complex: `()` matches a method that takes no parameters, whereas `(..)` matches any number of parameters (zero or more). The pattern `(*)` matches a method taking one parameter of any type, `(*,String)` matches a method taking two parameters, the first can be of any type, the second must be a String. Consult the Language Semantics section of the AspectJ Programming Guide for more information.

Some examples of common pointcut expressions are given below.

- the execution of any public method:

```
execution(public * *(..))
```

- the execution of any method with a name beginning with "set":

```
execution(* set*(..))
```

- the execution of any method defined by the `AccountService` interface:

```
execution(* com.xyz.service.AccountService.*(..))
```

- the execution of any method defined in the service package:

```
execution(* com.xyz.service.*.*(..))
```

- the execution of any method defined in the service package or a sub-package:

```
execution(* com.xyz.service..*.*(..))
```

- any join point (method execution only in Spring AOP) within the service package:

```
within(com.xyz.service.*)
```

- any join point (method execution only in Spring AOP) within the service package or a sub-package:

```
within(com.xyz.service..*)
```

- any join point (method execution only in Spring AOP) where the proxy implements the `AccountService` interface:

```
this(com.xyz.service.AccountService)
```

  'this' is more commonly used in a binding form :- see the following section on advice for how to make the proxy object available in the advice body.

- any join point (method execution only in Spring AOP) where the target object implements the `AccountService` interface:

```
target(com.xyz.service.AccountService)
```

  'target' is more commonly used in a binding form :- see the following section on advice for how to make the target object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the argument passed at runtime is `Serializable`:

```
args(java.io.Serializable)
```

  'args' is more commonly used in a binding form :- see the following section on advice for how to make the method arguments available in the advice body.

  Note that the pointcut given in this example is different to `execution(* *(java.io.Serializable))`: the args version matches if the argument passed at runtime is Serializable, the execution version matches if the method signature declares a single parameter of type `Serializable`.

- any join point (method execution only in Spring AOP) where the target object has an `@Transactional` annotation:

```
@target(org.springframework.transaction.annotation.Transactional)
```

  '@target' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the declared type of the target object has an `@Transactional` annotation:

```
@within(org.springframework.transaction.annotation.Transactional)
```

'@within' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) where the executing method has an `@Transactional` annotation:

```
@annotation(org.springframework.transaction.annotation.Transactional)
```

'@annotation' can also be used in a binding form :- see the following section on advice for how to make the annotation object available in the advice body.

- any join point (method execution only in Spring AOP) which takes a single parameter, and where the runtime type of the argument passed has the `@Classified` annotation:

```
@args(com.xyz.security.Classified)
```

'@args' can also be used in a binding form :- see the following section on advice for how to make the annotation object(s) available in the advice body.

6.2.4. Declaring advice

Advice is associated with a pointcut expression, and runs before, after, or around method executions matched by the pointcut. The pointcut expression may be either a simple reference to a named pointcut, or a pointcut expression declared in place.

6.2.4.1. Before advice

Before advice is declared in an aspect using the `@Before` annotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

If using an in-place pointcut expression we could rewrite the above example as:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class BeforeExample {

  @Before("execution(* com.xyz.myapp.dao.*.*(..))")
  public void doAccessCheck() {
    // ...
  }

}
```

6.2.4.2. After returning advice

After returning advice runs when a matched method execution returns normally. It is declared using the `@AfterReturning` annotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doAccessCheck() {
    // ...
  }

}
```

Note: it is of course possible to have multiple advice declarations, and other members as well, all inside the same aspect. We're just showing a single advice declaration in these examples to focus on the issue under discussion at the time.

Sometimes you need access in the advice body to the actual value that was returned. You can use the form of `@AfterReturning` that binds the return value for this:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterReturning;

@Aspect
public class AfterReturningExample {

  @AfterReturning(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    returning="retVal")
  public void doAccessCheck(Object retVal) {
    // ...
  }

}
```

The name used in the `returning` attribute must correspond to the name of a parameter in the advice method. When a method execution returns, the return value will be passed to the advice method as the corresponding argument value. A `returning` clause also restricts matching to only those method executions that return a value of the specified type (`Object` in this case, which will match any return value).

Please note that it is *not* possible to return a totally different reference when using after-returning advice.

6.2.4.3. After throwing advice

After throwing advice runs when a matched method execution exits by throwing an exception. It is declared using the `@AfterThrowing` annotation:

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doRecoveryActions() {
    // ...
  }

}
```

Often you want the advice to run only when exceptions of a given type are thrown, and you also often need access to the thrown exception in the advice body. Use the `throwing` attribute to both restrict matching (if desired, use `Throwable` as the exception type otherwise) and bind the thrown exception to an advice parameter.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.AfterThrowing;

@Aspect
public class AfterThrowingExample {

  @AfterThrowing(
    pointcut="com.xyz.myapp.SystemArchitecture.dataAccessOperation()",
    throwing="ex")
  public void doRecoveryActions(DataAccessException ex) {
    // ...
  }

}
```

The name used in the `throwing` attribute must correspond to the name of a parameter in the advice method. When a method execution exits by throwing an exception, the exception will be passed to the advice method as the corresponding argument value. A `throwing` clause also restricts matching to only those method executions that throw an exception of the specified type (`DataAccessException` in this case).

6.2.4.4. After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the `@After` annotation. After advice must be prepared to handle both normal and exception return conditions. It is typically used for releasing resources, etc.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.After;

@Aspect
public class AfterFinallyExample {

  @After("com.xyz.myapp.SystemArchitecture.dataAccessOperation()")
  public void doReleaseLock() {
    // ...
  }

}
```

6.2.4.5. Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements (i.e. don't use around advice if simple before advice would do).

Around advice is declared using the `@Around` annotation. The first parameter of the advice method must be of type `ProceedingJoinPoint`. Within the body of the advice, calling `proceed()` on the `ProceedingJoinPoint` causes the underlying method to execute. The `proceed` method may also be called passing in an `Object[]` - the values in the array will be used as the arguments to the method execution when it proceeds.
The behavior of proceed when called with an `Object[]` is a little different than the behavior of proceed for around advice compiled by the AspectJ compiler. For around advice written using the traditional AspectJ language, the number of arguments passed to proceed must match the number of arguments passed to the around advice (not the number of arguments taken by the underlying join point), and the value passed to

proceed in a given argument position supplants the original value at the join point for the entity the value was bound to. (Don't worry if this doesn't make sense right now!) The approach taken by Spring is simpler and a better match to its proxy-based, execution only semantics. You only need to be aware of this difference if you compiling @AspectJ aspects written for Spring and using proceed with arguments with the AspectJ compiler and weaver. There is a way to write such aspects that is 100% compatible across both Spring AOP and AspectJ, and this is discussed in the following section on advice parameters.

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.ProceedingJoinPoint;

@Aspect
public class AroundExample {

  @Around("com.xyz.myapp.SystemArchitecture.businessService()")
  public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
  }

}
```

The value returned by the around advice will be the return value seen by the caller of the method. A simple caching aspect for example could return a value from a cache if it has one, and invoke proceed() if it does not. Note that proceed may be invoked once, many times, or not at all within the body of the around advice, all of these are quite legal.

6.2.4.6. Advice parameters

Spring 2.0 offers fully typed advice - meaning that you declare the parameters you need in the advice signature (as we saw for the returning and throwing examples above) rather than work with `Object[]` arrays all the time. We'll see how to make argument and other contextual values available to the advice body in a moment. First let's take a look at how to write generic advice that can find out about the method the advice is currently advising.

6.2.4.6.1. Access to the current `JoinPoint`

Any advice method may declare as its first parameter, a parameter of type `org.aspectj.lang.JoinPoint` (please note that around advice is *required* to declare a first parameter of type `ProceedingJoinPoint`, which is a subclass of `JoinPoint`. The `JoinPoint` interface provides a number of useful methods such as `getArgs()` (returns the method arguments), `getThis()` (returns the proxy object), `getTarget()` (returns the target object), `getSignature()` (returns a description of the method that is being advised) and `toString()` (prints a useful description of the method being advised). Please do consult the Javadocs for full details.

6.2.4.6.2. Passing parameters to advice

We've already seen how to bind the returned value or exception value (using after returning and after throwing advice). To make argument values available to the advice body, you can use the binding form of `args`. If a parameter name is used in place of a type name in an args expression, then the value of the corresponding argument will be passed as the parameter value when the advice is invoked. An example should make this clearer. Suppose you want to advise the execution of dao operations that take an Account object as the first parameter, and you need access to the account in the advice body. You could write the following:

```
@Before("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
        "args(account,..)")
public void validateAccount(Account account) {
  // ...
}
```

The `args(account,..)` part of the pointcut expression serves two purposes: firstly, it restricts matching to only those method executions where the method takes at least one parameter, and the argument passed to that parameter is an instance of `Account`; secondly, it makes the actual `Account` object available to the advice via the `account` parameter.

Another way of writing this is to declare a pointcut that "provides" the `Account` object value when it matches a join point, and then just refer to the named pointcut from the advice. This would look as follows:

```
@Pointcut("com.xyz.myapp.SystemArchitecture.dataAccessOperation() &&" +
          "args(account,..)")
private void accountDataAccessOperation(Account account) {}

@Before("accountDataAccessOperation(account)")
public void validateAccount(Account account) {
  // ...
}
```

The interested reader is once more referred to the AspectJ programming guide for more details.

The proxy object (`this`), target object (`target`), and annotations (`@within, @target, @annotation, @args`) can all be bound in a similar fashion. The following example shows how you could match the execution of methods annotated with an `@Auditable` annotation, and extract the audit code.

First the definition of the `@Auditable` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Auditable {
        AuditCode value();
}
```

And then the advice that matches the execution of `@Auditable` methods:

```
@Before("com.xyz.lib.Pointcuts.anyPublicMethod() && " +
        "@annotation(auditable)")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

6.2.4.6.3. Determining argument names

The parameter binding in advice invocations relies on matching names used in pointcut expressions to declared parameter names in (advice and pointcut) method signatures. Parameter names are *not* available through Java reflection, so Spring AOP uses the following strategies to determine parameter names:

1. If the parameter names have been specified by the user explicitly, then the specified parameter names are used: both the advice and the pointcut annotations have an optional "argNames" attribute which can be used to specify the argument names of the annotated method - these argument names *are* available at runtime. For example:

```
@Before(
   value="com.xyz.lib.Pointcuts.anyPublicMethod() && @annotation(auditable)",
   argNames="auditable")
public void audit(Auditable auditable) {
  AuditCode code = auditable.value();
  // ...
}
```

   If an @AspectJ aspect has been compiled by the AspectJ compiler (ajc) then there is no need to add the

`argNames` attribute as the compiler will do this automatically.

2. Using the `'argNames'` attribute is a little clumsy, so if the `'argNames'` attribute has not been specified, then Spring AOP will look at the debug information for the class and try to determine the parameter names from the local variable table. This information will be present as long as the classes have been compiled with debug information (`'-g:vars'` at a minimum). The consequences of compiling with this flag on are: (1) your code will be slightly easier to understand (reverse engineer), (2) the class file sizes will be very slightly bigger (typically inconsequential), (3) the optimization to remove unused local variables will not be applied by your compiler. In other words, you should encounter no difficulties building with this flag on.

3. If the code has been compiled without the necessary debug information, then Spring AOP will attempt to deduce the pairing of binding variables to parameters (for example, if only one variable is bound in the pointcut expression, and the advice method only takes one parameter, the pairing is obvious!). If the binding of variables is ambiguous given the available information, then an `AmbiguousBindingException` will be thrown.

4. If all of the above strategies fail then an `IllegalArgumentException` will be thrown.

6.2.4.6.4. Proceeding with arguments

We remarked earlier that we would describe how to write a proceed call *with arguments* that works consistently across Spring AOP and AspectJ. The solution is simply to ensure that the advice signature binds each of the method parameters in order. For example:

```
@Around("execution(List<Account> find*(..)) &&" +
        "com.xyz.myapp.SystemArchitecture.inDataAccessLayer() && " +
        "args(accountHolderNamePattern)")
public Object preProcessQueryPattern(ProceedingJoinPoint pjp, String accountHolderNamePattern)
throws Throwable {
  String newPattern = preProcess(accountHolderNamePattern);
  return pjp.proceed(new Object[] {newPattern});
}
```

In many cases you will be doing this binding anyway (as in the example above).

6.2.4.7. Advice ordering

What happens when multiple pieces of advice all want to run at the same join point? Spring AOP follows the same precedence rules as AspectJ to determine the order of advice execution. The highest precedence advice runs first "on the way in" (so given two pieces of before advice, the one with highest precedence runs first). "On the way out" from a join point, the highest precedence advice runs last (so given two pieces of after advice, the one with the highest precedence will run second).

When two pieces of advice defined in *different* aspects both need to run at the same join point, unless you specify otherwise the order of execution is undefined. You can control the order of execution by specifying precedence. This is done in the normal Spring way by either implementing the `org.springframework.core.Ordered` interface in the aspect class or annotating it with the `Order` annotation. Given two aspects, the aspect returning the lower value from `Ordered.getValue()` (or the annotation value) has the higher precedence.

When two pieces of advice defined in *the same* aspect both need to run at the same join point, the ordering is undefined (since there is no way to retrieve the declaration order via reflection for javac-compiled classes). Consider collapsing such advice methods into one advice method per joinpoint in each aspect class, or refactor the pieces of advice into separate aspect classes - which can be ordered at the aspect level.

6.2.5. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the `@DeclareParents` annotation. This annotation is used to declare that matching types have a new parent (hence the name). For example, given an interface `UsageTracked`, and an implementation of that interface `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics via JMX for example.)

```
@Aspect
public class UsageTracking {

  @DeclareParents(value="com.xzy.myapp.service.*+",
                  defaultImpl=DefaultUsageTracked.class)
  public static UsageTracked mixin;

  @Before("com.xyz.myapp.SystemArchitecture.businessService() &&" +
          "this(usageTracked)")
  public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
  }

}
```

The interface to be implemented is determined by the type of the annotated field. The `value` attribute of the `@DeclareParents` annotation is an AspectJ type pattern :- any bean of a matching type will implement the UsageTracked interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the `UsageTracked` interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

6.2.6. Aspect instantiation models

(This is an advanced topic, so if you are just starting out with AOP you can safely skip it until later.)

By default there will be a single instance of each aspect within the application context. AspectJ calls this the singleton instantiation model. It is possible to define aspects with alternate lifecycles :- Spring supports AspectJ's `perthis` and `pertarget` instantiation models (`percflow`, `percflowbelow`, and `pertypewithin` are not currently supported).

A "perthis" aspect is declared by specifying a `perthis` clause in the `@Aspect` annotation. Let's look at an example, and then we'll explain how it works.

```
@Aspect("perthis(com.xyz.myapp.SystemArchitecture.businessService())")
public class MyAspect {

  private int someState;

  @Before(com.xyz.myapp.SystemArchitecture.businessService())
  public void recordServiceUsage() {
    // ...
  }

}
```

The effect of the `'perthis'` clause is that one aspect instance will be created for each unique service object executing a business service (each unique object bound to 'this' at join points matched by the pointcut

expression). The aspect instance is created the first time that a method is invoked on the service object. The aspect goes out of scope when the service object goes out of scope. Before the aspect instance is created, none of the advice within it executes. As soon as the aspect instance has been created, the advice declared within it will execute at matched join points, but only when the service object is the one this aspect is associated with. See the AspectJ programming guide for more information on per-clauses.

The `'pertarget'` instantiation model works in exactly the same way as perthis, but creates one aspect instance for each unique target object at matched join points.

6.2.7. Example

Now that you have seen how all the constituent parts work, let's put them together to do something useful!

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely to succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a `PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we will need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks:

```
@Aspect
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
      this.order = order;
   }

   @Around("com.xyz.myapp.SystemArchitecture.businessService()")
   public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
      int numAttempts = 0;
      PessimisticLockingFailureException lockFailureException;
      do {
         numAttempts++;
         try {
            return pjp.proceed();
         }
         catch(PessimisticLockingFailureException ex) {
            lockFailureException = ex;
         }
      }
      while(numAttempts <= this.maxRetries);
      throw lockFailureException;
   }

}
```

Note that the aspect implements the `Ordered` interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties will both be configured by Spring. The main action happens in the `doConcurrentOperation` around advice.

Notice that for the moment we're applying the retry logic to all `businessService()`s. We try to proceed, and if we fail with an `PessimisticLockingFailureException` we simply try again unless we have exhausted all of our retry attempts.

The corresponding Spring configuration is:

```
<aop:aspectj-autoproxy/>

<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
     <property name="maxRetries" value="3"/>
     <property name="order" value="100"/>
</bean>
```

To refine the aspect so that it only retries idempotent operations, we might define an `Idempotent` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and use the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only `@Idempotent` operations match:

```
@Around("com.xyz.myapp.SystemArchitecture.businessService() && " +
        "@annotation(com.xyz.myapp.service.Idempotent)")
public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
  ...
}
```

6.3. Schema-based AOP support

If you are unable to use Java 5, or simply prefer an XML-based format, then Spring 2.0 also offers support for defining aspects using the new "aop" namespace tags. The exact same pointcut expressions and advice kinds are supported as when using the @AspectJ style, hence in this section we will focus on the new *syntax* and refer the reader to the discussion in the previous section (Section 6.2, “@AspectJ support”) for an understanding of writing pointcut expressions and the binding of advice parameters.

To use the aop namespace tags described in this section, you need to import the spring-aop schema as described in Appendix A, *XML Schema-based configuration*. See Section A.2.6, “The `aop` schema” for how to import the tags in the aop namespace.

Within your Spring configurations, all aspect and advisor elements must be placed within an `<aop:config>` element (you can have more than one `<aop:config>` element in an application context configuration). An `<aop:config>` element can contain pointcut, advisor, and aspect elements (note these must be declared in that order).

Warning

The `<aop:config>` style of configuration makes heavy use of Spring's auto-proxying mechanism. This can cause issues (such as advice not being woven) if you are already using explicit auto-proxying via the use of `BeanNameAutoProxyCreator` or suchlike. The recommended usage pattern is to use either just the `<aop:config>` style, or just the `AutoProxyCreator` style.

6.3.1. Declaring an aspect

Using the schema support, an aspect is simply a regular Java object defined as a bean in your Spring application context. The state and behavior is captured in the fields and methods of the object, and the pointcut and advice information is captured in the XML.

An aspect is declared using the <aop:aspect> element, and the backing bean is referenced using the `ref` attribute:

```
<aop:config>
  <aop:aspect id="myAspect" ref="aBean">
    ...
  </aop:aspect>
</aop:config>

<bean id="aBean" class="...">
  ...
</bean>
```

The bean backing the aspect ("`aBean`" in this case) can of course be configured and dependency injected just like any other Spring bean.

6.3.2. Declaring a pointcut

A named pointcut can be declared inside an <aop:config> element, enabling the pointcut definition to be shared across several aspects and advisors.

A pointcut representing the execution of any business service in the service layer could be defined as follows:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

</aop:config>
```

Note that the pointcut expression itself is using the same AspectJ pointcut expression language as described in Section 6.2, “@AspectJ support”. If you are using the schema based declaration style with Java 5, you can refer to named pointcuts defined in types (@Aspects) within the pointcut expression, but this feature is not available on JDK 1.4 and below (it relies on the Java 5 specific AspectJ reflection APIs). On JDK 1.5 therefore, another way of defining the above pointcut would be:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="com.xyz.myapp.SystemArchitecture.businessService()"/>

</aop:config>
```

Assuming you have a `SystemArchitecture` aspect as described in Section 6.2.3.3, “Sharing common pointcut definitions”.

Declaring a pointcut inside an aspect is very similar to declaring a top-level pointcut:

```
<aop:config>

  <aop:aspect id="myAspect" ref="aBean">

    <aop:pointcut id="businessService"
          expression="execution(* com.xyz.myapp.service.*.*(..))"/>
```

```
    ...

  </aop:aspect>

</aop:config>
```

When combining pointcut sub-expressions, '&&' is awkward within an XML document, and so the keywords 'and', 'or' and 'not' can be used in place of '&&', '||' and '!' respectively.

Note that pointcuts defined in this way are referred to by their XML id, and cannot define pointcut parameters. The named pointcut support in the schema based definition style is thus more limited than that offered by the @AspectJ style.

6.3.3. Declaring advice

The same five advice kinds are supported as for the @AspectJ style, and they have exactly the same semantics.

6.3.3.1. Before advice

Before advice runs before a matched method execution. It is declared inside an `<aop:aspect>` using the <aop:before> element.

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

Here `dataAccessOperation` is the id of a pointcut defined at the top (`<aop:config>`) level. To define the pointcut inline instead, replace the `pointcut-ref` attribute with a `pointcut` attribute:

```
<aop:aspect id="beforeExample" ref="aBean">

    <aop:before
      pointcut="execution(* com.xyz.myapp.dao.*.*(..))"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

As we noted in the discussion of the @AspectJ style, using named pointcuts can significantly improve the readability of your code.

The method attribute identifies a method (`doAccessCheck`) that provides the body of the advice. This method must be defined for the bean referenced by the aspect element containing the advice. Before a data access operation is executed (a method execution join point matched by the pointcut expression), the "doAccessCheck" method on the aspect bean will be invoked.

6.3.3.2. After returning advice

After returning advice runs when a matched method execution completes normally. It is declared inside an `<aop:aspect>` in the same way as before advice. For example:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the return value within the advice body. Use the returning attribute to specify the name of the parameter to which the return value should be passed:

```
<aop:aspect id="afterReturningExample" ref="aBean">

    <aop:after-returning
      pointcut-ref="dataAccessOperation"
      returning="retVal"
      method="doAccessCheck"/>

    ...

</aop:aspect>
```

The doAccessCheck method must declare a parameter named `retVal`. The type of this parameter constrains matching in the same way as described for @AfterReturning. For example, the method signature may be declared as:

```
public void doAccessCheck(Object retVal) {...
```

6.3.3.3. After throwing advice

After throwing advice executes when a matched method execution exits by throwing an exception. It is declared inside an `<aop:aspect>` using the after-throwing element:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      method="doRecoveryActions"/>

    ...

</aop:aspect>
```

Just as in the @AspectJ style, it is possible to get hold of the thrown exception within the advice body. Use the throwing attribute to specify the name of the parameter to which the exception should be passed:

```
<aop:aspect id="afterThrowingExample" ref="aBean">

    <aop:after-throwing
      pointcut-ref="dataAccessOperation"
      throwing="dataAccessEx"
      method="doRecoveryActions"/>

    ...

</aop:aspect>
```

The doRecoveryActions method must declare a parameter named `dataAccessEx`. The type of this parameter constrains matching in the same way as described for @AfterThrowing. For example, the method signature may be declared as:

```
public void doRecoveryActions(DataAccessException dataAccessEx) {...
```

6.3.3.4. After (finally) advice

After (finally) advice runs however a matched method execution exits. It is declared using the `after` element:

```
<aop:aspect id="afterFinallyExample" ref="aBean">

    <aop:after
      pointcut-ref="dataAccessOperation"
      method="doReleaseLock"/>

    ...

</aop:aspect>
```

6.3.3.5. Around advice

The final kind of advice is around advice. Around advice runs "around" a matched method execution. It has the opportunity to do work both before and after the method executes, and to determine when, how, and even if, the method actually gets to execute at all. Around advice is often used if you need to share state before and after a method execution in a thread-safe manner (starting and stopping a timer for example). Always use the least powerful form of advice that meets your requirements; don't use around advice if simple before advice would do.

Around advice is declared using the `aop:around` element. The first parameter of the advice method must be of type `ProceedingJoinPoint`. Within the body of the advice, calling `proceed()` on the `ProceedingJoinPoint` causes the underlying method to execute. The `proceed` method may also be calling passing in an `Object[]` - the values in the array will be used as the arguments to the method execution when it proceeds. See Section 6.2.4.5, “Around advice” for notes on calling proceed with an `Object[]`.

```
<aop:aspect id="aroundExample" ref="aBean">

    <aop:around
      pointcut-ref="businessService"
      method="doBasicProfiling"/>

    ...

</aop:aspect>
```

The implementation of the `doBasicProfiling` advice would be exactly the same as in the @AspectJ example (minus the annotation of course):

```
public Object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable {
    // start stopwatch
    Object retVal = pjp.proceed();
    // stop stopwatch
    return retVal;
}
```

6.3.3.6. Advice parameters

The schema based declaration style supports fully typed advice in the same way as described for the @AspectJ support - by matching pointcut parameters by name against advice method parameters. See Section 6.2.4.6, “Advice parameters” for details.

If you wish to explicity specify argument names for the advice methods (not relying on either of the detection

strategies previously described) then this is done using the `arg-names` attribute of the advice element. For example:

```
<aop:before
  pointcut="com.xyz.lib.Pointcuts.anyPublicMethod() and @annotation(auditable)"
  method="audit"
  arg-names="auditable"/>
```

The `arg-names` attribute accepts a comma-delimited list of parameter names.

Find below a slightly more involved example of the XSD-based approach that illustrates some around advice used in conjunction with a number of strongly typed parameters.

```
package x.y.service;

public interface FooService {

   Foo getFoo(String fooName, int age);
}

public class DefaultFooService implements FooService {

   public Foo getFoo(String name, int age) {
      return new Foo(name, age);
   }
}
```

Next up is the aspect. Notice the fact that the `profile(..)` method accepts a number of strongly-typed parameters, the first of which happens to be the join point used to proceed with the method call: the presence of this parameter is an indication that the `profile(..)` is to be used as `around` advice:

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;

public class SimpleProfiler {

   public Object profile(ProceedingJoinPoint call, String name, int age) throws Throwable {
      StopWatch clock = new StopWatch(
            "Profiling for '" + name + "' and '" + age + "'");
      try {
         clock.start(call.toShortString());
         return call.proceed();
      } finally {
         clock.stop();
         System.out.println(clock.prettyPrint());
      }
   }
}
```

Finally, here is the XML configuration that is required to effect the execution of the above advice for a particular joinpoint:

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

   <!-- this is the object that will be proxied by Spring's AOP infrastructure -->
   <bean id="fooService" class="x.y.service.DefaultFooService"/>

   <!-- this is the actual advice itself -->
   <bean id="profiler" class="x.y.SimpleProfiler"/>

   <aop:config>
```

```
      <aop:aspect ref="profiler">

         <aop:pointcut id="theExecutionOfSomeFooServiceMethod"
                    expression="execution(* x.y.service.FooService.getFoo(String,int))
                    and args(name, age)"/>

         <aop:around pointcut-ref="theExecutionOfSomeFooServiceMethod"
                  method="profile"/>

      </aop:aspect>
   </aop:config>

</beans>
```

If we had the following driver script, we would get output something like this on standard output:

```
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import x.y.service.FooService;

public final class Boot {

   public static void main(final String[] args) throws Exception {
      BeanFactory ctx = new ClassPathXmlApplicationContext("x/y/plain.xml");
      FooService foo = (FooService) ctx.getBean("fooService");
      foo.getFoo("Pengo", 12);
   }
}
```

```
StopWatch 'Profiling for 'Pengo' and '12'': running time (millis) = 0
-----------------------------------------
ms     %     Task name
-----------------------------------------
00000  ?  execution(getFoo)
```

6.3.3.7. Advice ordering

When multiple advice needs to execute at the same join point (executing method) the ordering rules are as described in Section 6.2.4.7, “Advice ordering”. The precedence between aspects is determined by either adding the `Order` annotation to the bean backing the aspect or by having the bean implement the `Ordered` interface.

6.3.4. Introductions

Introductions (known as inter-type declarations in AspectJ) enable an aspect to declare that advised objects implement a given interface, and to provide an implementation of that interface on behalf of those objects.

An introduction is made using the `aop:declare-parents` element inside an `aop:aspect` This element is used to declare that matching types have a new parent (hence the name). For example, given an interface `UsageTracked`, and an implementation of that interface `DefaultUsageTracked`, the following aspect declares that all implementors of service interfaces also implement the `UsageTracked` interface. (In order to expose statistics via JMX for example.)

```
<aop:aspect id="usageTrackerAspect" ref="usageTracking">

  <aop:declare-parents
      types-matching="com.xzy.myapp.service.*+"
      implement-interface="com.xyz.myapp.service.tracking.UsageTracked"
      default-impl="com.xyz.myapp.service.tracking.DefaultUsageTracked"/>

  <aop:before
    pointcut="com.xyz.myapp.SystemArchitecture.businessService()
              and this(usageTracked)"
    method="recordUsage"/>
```

```
</aop:aspect>
```

The class backing the `usageTracking` bean would contain the method:

```
public void recordUsage(UsageTracked usageTracked) {
    usageTracked.incrementUseCount();
}
```

The interface to be implemented is determined by `implement-interface` attribute. The value of the `types-matching` attribute is an AspectJ type pattern :- any bean of a matching type will implement the `UsageTracked` interface. Note that in the before advice of the above example, service beans can be directly used as implementations of the `UsageTracked` interface. If accessing a bean programmatically you would write the following:

```
UsageTracked usageTracked = (UsageTracked) context.getBean("myService");
```

6.3.5. Aspect instantiation models

The only supported instantiation model for schema-defined aspects is the singleton model. Other instantiation models may be supported in future releases.

6.3.6. Advisors

The concept of "advisors" is brought forward from the AOP support defined in Spring 1.2 and does not have a direct equivalent in AspectJ. An advisor is like a small self-contained aspect that has a single piece of advice. The advice itself is represented by a bean, and must implement one of the advice interfaces described in Section 7.3.2, “Advice types in Spring”. Advisors can take advantage of AspectJ pointcut expressions though.

Spring 2.0 supports the advisor concept with the `<aop:advisor>` element. You will most commonly see it used in conjunction with transactional advice, which also has its own namespace support in Spring 2.0. Here's how it looks:

```
<aop:config>

  <aop:pointcut id="businessService"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

  <aop:advisor
      pointcut-ref="businessService"
      advice-ref="tx-advice"/>

</aop:config>

<tx:advice id="tx-advice">
  <tx:attributes>
    <tx:method name="*" propagation="REQUIRED"/>
  </tx:attributes>
</tx:advice>
```

As well as the `pointcut-ref` attribute used in the above example, you can also use the `pointcut` attribute to define a pointcut expression inline.

To define the precedence of an advisor so that the advice can participate in ordering, use the `order` attribute to define the `Ordered` value of the advisor.

6.3.7. Example

Let's see how the concurrent locking failure retry example from Section 6.2.7, “Example” looks when rewritten using the schema support.

The execution of business services can sometimes fail due to concurrency issues (for example, deadlock loser). If the operation is retried, it is quite likely it will succeed next time round. For business services where it is appropriate to retry in such conditions (idempotent operations that don't need to go back to the user for conflict resolution), we'd like to transparently retry the operation to avoid the client seeing a `PessimisticLockingFailureException`. This is a requirement that clearly cuts across multiple services in the service layer, and hence is ideal for implementing via an aspect.

Because we want to retry the operation, we'll need to use around advice so that we can call proceed multiple times. Here's how the basic aspect implementation looks (it's just a regular Java class using the schema support):

```
public class ConcurrentOperationExecutor implements Ordered {

   private static final int DEFAULT_MAX_RETRIES = 2;

   private int maxRetries = DEFAULT_MAX_RETRIES;
   private int order = 1;

   public void setMaxRetries(int maxRetries) {
      this.maxRetries = maxRetries;
   }

   public int getOrder() {
      return this.order;
   }

   public void setOrder(int order) {
      this.order = order;
   }

   public Object doConcurrentOperation(ProceedingJoinPoint pjp) throws Throwable {
      int numAttempts = 0;
      PessimisticLockingFailureException lockFailureException;
      do {
         numAttempts++;
         try {
            return pjp.proceed();
         }
         catch(PessimisticLockingFailureException ex) {
            lockFailureException = ex;
         }
      }
      while(numAttempts <= this.maxRetries);
      throw lockFailureException;
   }

}
```

Note that the aspect implements the `Ordered` interface so we can set the precedence of the aspect higher than the transaction advice (we want a fresh transaction each time we retry). The `maxRetries` and `order` properties will both be configured by Spring. The main action happens in the `doConcurrentOperation` around advice method. We try to proceed, and if we fail with a `PessimisticLockingFailureException` we simply try again unless we have exhausted all of our retry attempts.
This class is identical to the one used in the @AspectJ example, but with the annotations removed.

The corresponding Spring configuration is:

```
<aop:config>

  <aop:aspect id="concurrentOperationRetry" ref="concurrentOperationExecutor">
```

```
    <aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..))"/>

    <aop:around
       pointcut-ref="idempotentOperation"
       method="doConcurrentOperation"/>

  </aop:aspect>

</aop:config>

<bean id="concurrentOperationExecutor"
  class="com.xyz.myapp.service.impl.ConcurrentOperationExecutor">
     <property name="maxRetries" value="3"/>
     <property name="order" value="100"/>
</bean>
```

Notice that for the time being we assume that all business services are idempotent. If this is not the case we can refine the aspect so that it only retries genuinely idempotent operations, by introducing an `Idempotent` annotation:

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Idempotent {
  // marker annotation
}
```

and using the annotation to annotate the implementation of service operations. The change to the aspect to only retry idempotent operations simply involves refining the pointcut expression so that only `@Idempotent` operations match:

```
  <aop:pointcut id="idempotentOperation"
        expression="execution(* com.xyz.myapp.service.*.*(..)) and
                    @annotation(com.xyz.myapp.service.Idempotent)"/>
```

6.4. Choosing which AOP declaration style to use

Once you have decided that an aspect is the best approach for implementing a given requirement, how do you decide between using Spring AOP or AspectJ, and between the Aspect language (code) style, @AspectJ annotation style, and the XML style? These decisions are influenced by a number of factors including application requirements, development tools, and team familiarity with AOP.

6.4.1. Spring AOP or full AspectJ?

Use the simplest thing that can work. Spring AOP is simpler than using full AspectJ as there is no requirement to introduce the AspectJ compiler / weaver into your development and build processes. If you only need to advise the execution of operations on Spring beans, then Spring AOP is the right choice. If you need to advise domain objects, or any other object not managed by the Spring container, then you will need to use AspectJ. You will also need to use AspectJ if you wish to advise join points other than simple method executions (for example, call join points, field get or set join points, and so on).

When using AspectJ, you have the choice of the AspectJ language syntax (also known as the "code style") or the @AspectJ annotation style. If aspects play a large role in your design, and you are able to use the AspectJ Development Tools (AJDT) in Eclipse, then the AspectJ language syntax is the preferred option: it is cleaner and simpler because the language was purposefully designed for writing aspects. If you are not using Eclipse, or have only a few aspects that do not play a major role in your application, then you may want to consider using the @AspectJ style and sticking with a regular Java compilation in your IDE, and adding an aspect

weaving (linking) phase to your build scripts.

6.4.2. @AspectJ or XML for Spring AOP?

The XML style will be most familiar to existing Spring users. It can be used with any JDK level (referring to named pointcuts from within pointcut expressions does still require Java 5 though) and is backed by genuine POJOs. When using AOP as a tool to configure enterprise services (a good test is whether you consider the pointcut expression to be a part of your configuration you might want to change independently) then XML can be a good choice. With the XML style it is arguably clearer from your configuration what aspects are present in the system.

The XML style has two disadvantages. Firstly it does not fully encapsulate the implementation of the requirement it addresses in a single place. The DRY principle says that there should be a single, unambiguous, authoritative representation of any piece of knowledge within a system. When using the XML style, the knowledge of how a requirement is implemented is split across the declaration of the backing bean class, and the XML in the configuration file. When using the @AspectJ style there is a single module - the aspect - in which this information is encapsulated. Secondly, the XML style is more limited in what in can express than the @AspectJ style: only the "singleton" aspect instantiation model is supported, and it is not possible to combine named pointcuts declared in XML. For example, in the @AspectJ style we can write something like:

```
@Pointcut(execution(* get*()))
public void propertyAccess() {}

@Pointcut(execution(org.xyz.Account+ *(..))
public void operationReturningAnAccount() {}

@Pointcut(propertyAccess() && operationReturningAnAccount())
public void accountPropertyAccess() {}
```

In the XML style I certainly can declare the first two pointcuts:

```
<aop:pointcut id="propertyAccess"
    expression="execution(* get*())"/>

<aop:pointcut id="operationReturningAnAccount"
    expression="execution(org.xyz.Account+ *(..))"/>
```

The downside of the XML approach becomes evident in this case because I cannot define the '`accountPropertyAccess`' pointcut by combining these definitions.

The @AspectJ style supports additional instantiation models, and richer pointcut composition. It has the advantage of keeping the aspect as a modular unit. It also has the advantage the @AspectJ aspects can be understood both by Spring AOP and by AspectJ - so if you later decide you need the capabilities of AspectJ to implement additional requirements then it is very easy to migrate to an AspectJ based approach.

So much for the pros and cons of each style then: which is best? If you are not using Java 5 (or above) then clearly the XML-style is appropriate because it is the only option available to you. If you are using Java 5+, then you really will have to come to your own decision as to which style suits you best. In the experience of the Spring team, we advocate the use of the @AspectJ style whenever there are aspects that do more than simple "configuration" of enterprise services. If you are writing, have written, or have access to an aspect that is not part of the business contract of a particular class (such as a tracing aspect), then the XML-style is better.

6.5. Mixing aspect types

It is perfectly possible to mix @AspectJ style aspects using the autoproxying support, schema-defined

<aop:aspect> aspects, <aop:advisor> declared advisors and even proxies and interceptors defined using the Spring 1.2 style in the same configuration. All of these are implemented using the same underlying support mechanism and will co-exist without any difficulty.

6.6. Proxying mechanisms

Spring AOP uses either JDK dynamic proxies or CGLIB to create the proxy for a given target object. (JDK dynamic proxies are preferred whenever you have a choice).

If the target object to be proxied implements at least one interface then a JDK dynamic proxy will be used. All of the interfaces implemented by the target type will be proxied. If the target object does not implement any interfaces then a CGLIB proxy will be created.

If you want to force the use of CGLIB proxying (for example, to proxy every method defined for the target object, not just those implemented by its interfaces) you can do so. However, there are some issues to consider:

- `final` methods cannot be advised, as they cannot be overriden.
- You will need the CGLIB 2 binaries on your classpath, whereas dynamic proxies are available with the JDK. Spring will automatically warn you when it needs CGLIB and the CGLIB library classes are not found on the classpath.
- The constructor of your proxied object will be called twice. This is a natural consequence of the CGLIB proxy model whereby a subclass is generated for each proxied object. For each proxied instance, two objects are created: the actual proxied object and an instance of the subclass that implements the advice. This behavior is not exhibited when using JDK proxies. Usually, calling the constructor of the proxied type twice, is not an issue, as there are usually only assignments taking place and no real logic is implemented in the constructor.

To force the use of CGLIB proxies set the value of the `proxy-target-class` attribute of the <aop:config> element to true:

```
<aop:config proxy-target-class="true">
    <!-- other beans defined here... -->
</aop:config>
```

To force CGLIB proxying when using the @AspectJ autoproxy support, set the `'proxy-target-class'` attribute of the <aop:aspectj-autoproxy> element to `true`:

```
<aop:aspectj-autoproxy proxy-target-class="true"/>
```

Note

Multiple <aop:config/> sections are collapsed into a single unified auto-proxy creator at runtime, which applies the *strongest* proxy settings that any of the <aop:config/> sections (typically from different XML bean definition files) specified. This also applies to the <tx:annotation-driven/> and <aop:aspectj-autoproxy/> elements.

To be clear: using 'proxy-target-class="true"' on <tx:annotation-driven/>, <aop:aspectj-autoproxy/> or <aop:config/> elements will force the use of CGLIB proxies *for all three of them*.

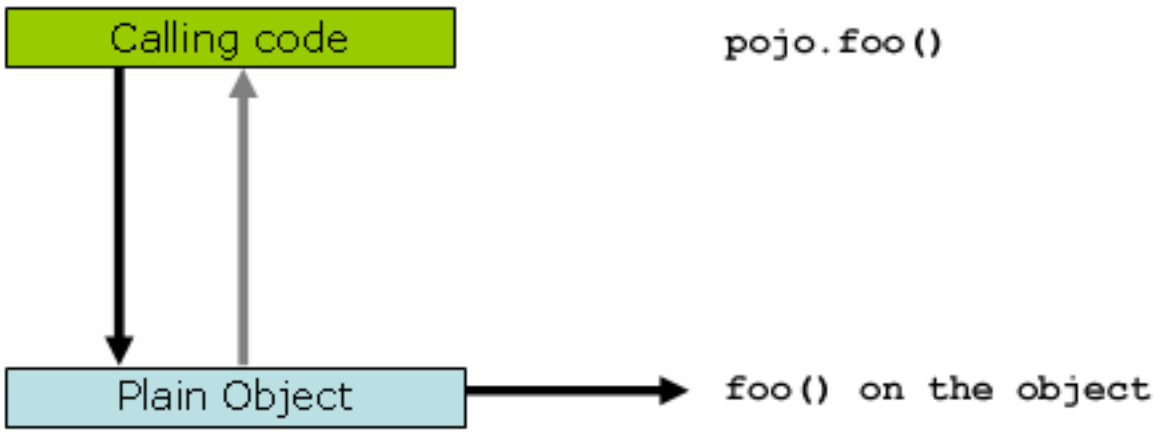
6.6.1. Understanding AOP proxies

Spring AOP is *proxy-based*. It is vitally important that you grasp the semantics of what that last statement actually means before you write your own aspects or use any of the Spring AOP-based aspects supplied with the Spring Framework.

Consider first the scenario where you have a plain-vanilla, un-proxied, nothing-special-about-it, straight object reference, as illustrated by the following code snippet.

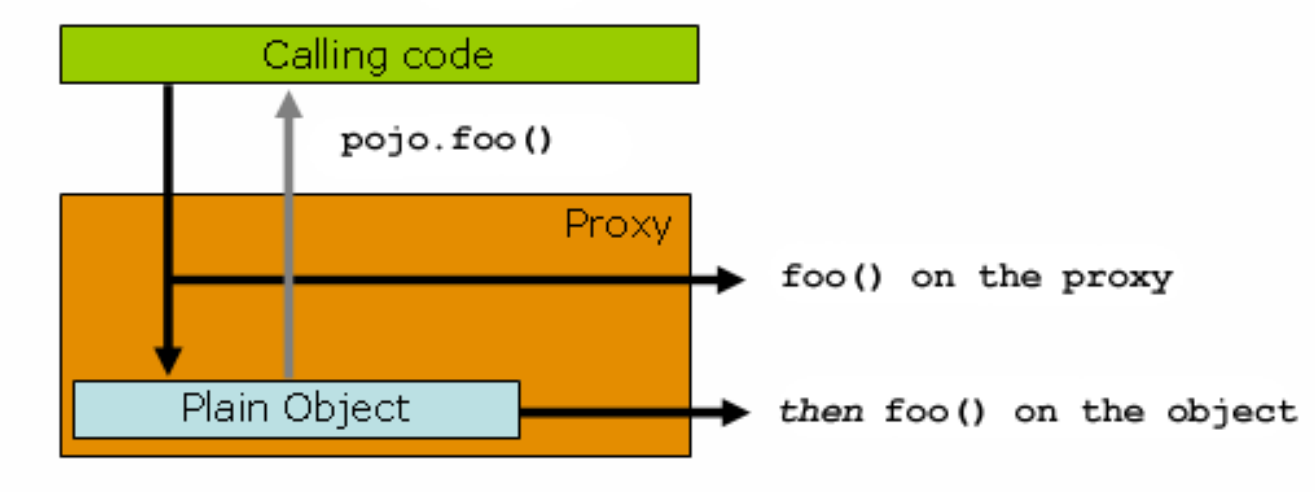
```
public class SimplePojo implements Pojo {

   public void foo() {
      // this next method invocation is a direct call on the 'this' reference
      this.bar();
   }

   public void bar() {
      // some logic...
   }
}
```

If you invoke a method on an object reference, the method is invoked *directly* on that object reference, as can be seen below.




```
public class Main {

   public static void main(String[] args) {

      Pojo pojo = new SimplePojo();

      // this is a direct method call on the 'pojo' reference
      pojo.foo();
   }
}
```

Things change slightly when the reference that client code has is a proxy. Consider the following diagram and code snippet.




```
public class Main {

   public static void main(String[] args) {

      ProxyFactory factory = new ProxyFactory(new SimplePojo());
      factory.addInterface(Pojo.class);
      factory.addAdvice(new RetryAdvice());

      Pojo pojo = (Pojo) factory.getProxy();

      // this is a method call on the proxy!
      pojo.foo();
   }
}
```

The key thing to understand here is that the client code inside the `main(..)` of the `Main` class *has a reference to the proxy*. This means that method calls on that object reference will be calls on the proxy, and as such the proxy will be able to delegate to all of the interceptors (advice) that are relevant to that particular method call. However, once the call has finally reached the target object, the `SimplePojo` reference in this case, any method calls that it may make on itself, such as `this.bar()` or `this.foo()`, are going to be invoked against the `this` reference, and *not* the proxy. This has important implications. It means that self-invocation is *not* going to result in the advice associated with a method invocation getting a chance to execute.

Okay, so what is to be done about this? The best approach (the term best is used loosely here) is to refactor your code such that the self-invocation does not happen. For sure, this does entail some work on your part, but it is the best, least-invasive approach. The next approach is absolutely horrendous, and I am almost reticent to point it out precisely because it is so horrendous. You can (choke!) totally tie the logic within your class to Spring AOP by doing this:

```
public class SimplePojo implements Pojo {

   public void foo() {
      // this works, but... gah!
      ((Pojo) AopContext.currentProxy()).bar();
   }

   public void bar() {
      // some logic...
   }
}
```

This totally couples your code to Spring AOP, *and* it makes the class itself aware of the fact that it is being used in an AOP context, which flies in the face of AOP. It also requires some additional configuration when the proxy is being created:

```
public class Main {
```

```
   public static void main(String[] args) {

      ProxyFactory factory = new ProxyFactory(new SimplePojo());
      factory.adddInterface(Pojo.class);
      factory.addAdvice(new RetryAdvice());
      factory.setExposeProxy(true);

      Pojo pojo = (Pojo) factory.getProxy();

      // this is a method call on the proxy!
      pojo.foo();
   }
}
```

Finally, it must be noted that AspectJ does not have this self-invocation issue because it is not a proxy-based AOP framework.

6.7. Programmatic creation of @AspectJ Proxies

In addition to declaring aspects in your configuration using either `<aop:config>` or `<aop:aspectj-autoproxy>`, it is also possible programmatically to create proxies that advise target objects. For the full details of Spring's AOP API, see the next chapter. Here we want to focus on the ability to automatically create proxies using @AspectJ aspects.

The class `org.springframework.aop.aspectj.annotation.AspectJProxyFactory` can be used to create a proxy for a target object that is advised by one or more @AspectJ aspects. Basic usage for this class is very simple, as illustrated below. See the Javadocs for full information.

```
// create a factory that can generate a proxy for the given target object
AspectJProxyFactory factory = new AspectJProxyFactory(targetObject);

// add an aspect, the class must be an @AspectJ aspect
// you can call this as many times as you need with different aspects
factory.addAspect(SecurityManager.class);

// you can also add existing aspect instances, the type of the object supplied must be an @AspectJ aspect
factory.addAspect(usageTracker);

// now get the proxy object...
MyInterfaceType proxy = factory.getProxy();
```

6.8. Using AspectJ with Spring applications

Everything we've covered so far in this chapter is pure Spring AOP. In this section, we're going to look at how you can use the AspectJ compiler/weaver instead of or in addition to Spring AOP if your needs go beyond the facilities offered by Spring AOP alone.

Spring ships with a small AspectJ aspect library (it's available standalone in your distribution as `spring-aspects.jar`, you'll need to add this to your classpath to use the aspects in it). Section 6.8.1, “Using AspectJ to dependency inject domain objects with Spring” and Section 6.8.2, “Other Spring aspects for AspectJ” discuss the content of this library and how you can use it. Section 6.8.3, “Configuring AspectJ aspects using Spring IoC” discusses how to dependency inject AspectJ aspects that are woven using the AspectJ compiler. Finally, Section 6.8.4, “Using AspectJ Load-time weaving (LTW) with Spring applications” provides an introduction to load-time weaving for Spring applications using AspectJ.

6.8.1. Using AspectJ to dependency inject domain objects with Spring

The Spring container instantiates and configures beans defined in your application context. It is also possible to ask a bean factory to configure a *pre-existing* object given the name of a bean definition containing the configuration to be applied. The `spring-aspects.jar` contains an annotation-driven aspect that exploits this capability to allow dependency-injection of any object. The support is intended to be used for objects created *outside of the control of any container*. Domain objects often fall into this category: they may be created programmatically using the `new` operator, or by an ORM tool as a result of a database query.

The `@Configurable` annotation marks a class as eligible for Spring-driven configuration. In the simplest case it can be used just as a marker annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable
public class Account {
   ...
}
```

When used as a marker interface in this way, Spring will configure new instances of the annotated type (`Account` in this case) using a prototypical bean definition with the same name as the fully-qualified type name (`com.xyz.myapp.domain.Account`). Since the default name for a bean is the fully-qualified name of its type, a convenient way to declare the prototype definition is simply to omit the id attribute:

```
<bean class="com.xyz.myapp.domain.Account" scope="prototype">
  <property name="fundsTransferService" ref="fundsTransferService"/>
  ...
</bean>
```

If you want to explicitly specify the name of the prototype bean definition to use, you can do so directly in the annotation:

```
package com.xyz.myapp.domain;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("account")
public class Account {
   ...
}
```

Spring will now look for a bean definition named "account" and use that as a prototypical definition to configure new `Account` instances.

You can also use autowiring to avoid having to specify a prototypical bean definition at all. To have Spring apply autowiring use the autowire property of the `@Configurable` annotation: specify either `@Configurable(autowire=Autowire.BY_TYPE)` or `@Configurable(autowire=Autowire.BY_NAME` for autowiring by type or by name respectively.

Finally you can enable Spring dependency checking for the object references in the newly created and configured object by using the `dependencyCheck` attribute (for example: `@Configurable(autowire=Autowire.BY_NAME,dependencyCheck=true)`). If this attribute is set to true, then Spring will validate after configuration that all properties (*that are not primitives or collections*) have been set.

Using the annotation on its own does nothing of course. It's the `AnnotationBeanConfigurerAspect` in `spring-aspects.jar` that acts on the presence of the annotation. In essence the aspect says "after returning from the initialization of a new object of a type with the `@Configurable` annotation, configure the newly created object using Spring in accordance with the properties of the annotation". For this to work the annotated

types must be woven with the AspectJ weaver - you can either use a build-time ant or maven task to do this (see for example the AspectJ Development Environment Guide) or load-time weaving (see Section 6.8.4, “Using AspectJ Load-time weaving (LTW) with Spring applications”).

The `AnnotationBeanConfigurerAspect` itself needs configuring by Spring (in order to obtain a reference to the bean factory that is to be used to configure new objects). The Spring AOP namespace defines a convenient tag for doing this. Simply include the following in your application context configuration:

```
<aop:spring-configured/>
```

If you are using the DTD instead of schema, the equivalent definition is:

```
<bean
  class="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"
  factory-method="aspectOf"/>
```

Instances of @Configurable objects created *before* the aspect has been configured will result in a warning being issued to the log and no configuration of the object taking place. An example might be a bean in the Spring configuration that creates domain objects when it is initialized by Spring. In this case you can use the "depends-on" bean attribute to manually specify that the bean depends on the configuration aspect.

```
<bean id="myService"
  class="com.xzy.myapp.service.MyService"
  depends-on="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect">
  ...
</bean>
```

6.8.1.1. Unit testing `@Configurable` objects

One of the goals of the `@Configurable` support is to enable independent unit testing of domain objects without the difficulties associated with hard-coded lookups. If `@Configurable` types have not been woven by AspectJ then the annotation has no affect during unit testing, and you can simply set mock or stub property references in the object under test and proceed as normal. If `@Configurable` types *have* been woven by AspectJ then you can still unit test outside of the container as normal, but you will see a warning message each time that you construct an `@Configurable` object indicating that it has not been configured by Spring.

6.8.1.2. Working with multiple application contexts

The `AnnotationBeanConfigurerAspect` used to implement the `@Configurable` support is an AspectJ singleton aspect. The scope of a singleton aspect is the same as the scope of `static` members, that is to say there is one aspect instance per classloader that defines the type. This means that if you define multiple application contexts within the same classloader hierarchy you need to consider where to define the `<aop:spring-configured/>` bean and where to place `spring-aspects.jar`on the classpath.

Consider a typical Spring web-app configuration with a shared parent application context defining common business services and everything needed to support them, and one child application context per servlet containing definitions particular to that servlet. All of these contexts will co-exist within the same classloader hierarchy, and so the `AnnotationBeanConfigurerAspect` can only hold a reference to one of them. In this case we recommend defining the `<aop:spring-configured/>` bean in the shared (parent) application context: this defines the services that you are likely to want to inject into domain objects. A consequence is that you cannot configure domain objects with references to beans defined in the child (servlet-specific) contexts using the @Configurable mechanism (probably not something you want to do anyway!).

When deploying multiple web-apps within the same container, ensure that each web-application loads the types in `spring-aspects.jar` using its own classloader (for example, by placing `spring-aspects.jar` in

'WEB-INF/lib'). If spring-aspects.jar is only added to the container wide classpath (and hence loaded by the shared parent classloader), all web applications will share the same aspect instance which is probably not what you want.

6.8.2. Other Spring aspects for AspectJ

In addition to the @Configurable support, spring-aspects.jar contains an AspectJ aspect that can be used to drive Spring's transaction management for types and methods annotated with the @Transactional annotation. This is primarily intended for users who want to use Spring's transaction support outside of the Spring container.

The aspect that interprets @Transactional annotations is the AnnotationTransactionAspect. When using this aspect, you must annotate the *implementation* class (and/or methods within that class), *not* the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are *not inherited*.

A @Transactional annotation on a class specifies the default transaction semantics for the execution of any *public* operation in the class.

A @Transactional annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Methods with public, protected, and default visibility may all be annotated. Annotating protected and default visibility methods directly is the only way to get transaction demarcation for the execution of such operations.

For AspectJ programmers that want to use the Spring configuration and transaction management support but don't want to (or can't) use annotations, spring-aspects.jar also contains abstract aspects you can extend to provide your own pointcut definitions. See the Javadocs for AbstractBeanConfigurerAspect and AbstractTransactionAspect for more information. As an example, the following excerpt shows how you could write an aspect to configure all instances of objects defined in the domain model using prototypical bean definitions that match the fully-qualified class names:

```
public aspect DomainObjectConfiguration extends AbstractBeanConfigurerAspect {

  public DomainObjectConfiguration() {
    setBeanWiringInfoResolver(new ClassNameBeanWiringInfoResolver());
  }

  // the creation of a new bean (any object in the domain model)
  protected pointcut beanCreation(Object beanInstance) :
    initialization(new(..)) &&
    SystemArchitecture.inDomainModel() &&
    this(beanInstance);

}
```

6.8.3. Configuring AspectJ aspects using Spring IoC

When using AspectJ aspects with Spring applications, it's natural to want to configure such aspects using Spring. The AspectJ runtime itself is responsible for aspect creation, and the means of configuring the AspectJ created aspects via Spring depends on the AspectJ instantiation model (per-clause) used by the aspect.

The majority of AspectJ aspects are *singleton* aspects. Configuration of these aspects is very easy, simply create a bean definition referencing the aspect type as normal, and include the bean attribute 'factory-method="aspectOf"'. This ensures that Spring obtains the aspect instance by asking AspectJ for it rather than trying to create an instance itself. For example:

```
<bean id="profiler" class="com.xyz.profiler.Profiler"
      factory-method="aspectOf">
  <property name="profilingStrategy" ref="jamonProfilingStrategy"/>
```

```
</bean>
```

For non-singleton aspects, the easiest way to configure them is to create prototypical bean definitions and annotate use the @Configurable support from `spring-aspects.jar` to configure the aspect instances once they have bean created by the AspectJ runtime.

If you have some @AspectJ aspects that you want to weave with AspectJ (for example, using load-time weaving for domain model types) and other @AspectJ aspects that you want to use with Spring AOP, and these aspects are all configured using Spring, then you'll need to tell the Spring AOP @AspectJ autoproxying support which subset of the @AspectJ aspects defined in the configuration should be used for autoproxying. You can do this by using one or more `<include/>` elements inside the `<aop:aspectj-autoproxy/>` declaration. Each include element specifies a name pattern, and only beans with names matched by at least one of the patterns will be used for Spring AOP autoproxy configuration:

```
<aop:aspectj-autoproxy>
  <aop:include name="thisBean"/>
  <aop:include name="thatBean"/>
</aop:aspectj-autoproxy>
```

6.8.4. Using AspectJ Load-time weaving (LTW) with Spring applications

Load-time weaving (or LTW) refers to the process of weaving AspectJ aspects with an application's class files as they are loaded into the VM. For full details on configuring load-time weaving with AspectJ, see the LTW section of the AspectJ Development Environment Guide . We will focus here on the essentials of configuring load-time weaving for Spring applications running on Java 5.

Load-time weaving is controlled by defining a file '`aop.xml`' in the META-INF directory. AspectJ automatically looks for all 'META-INF/aop.xml' files visible on the classpath and configures itself based on the aggregation of their content.

A basic META-INF/aop.xml for your application should look like this:

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN"    "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
   <weaver>
         <include within="com.xyz.myapp..*"/>
   </weaver>
</aspectj>
```

The `<include/>` element tells AspectJ which set of types should be included in the weaving process. Use the package prefix for your application followed by "..*" (meaning '... and any type defined in a subpackage of this') as a good default. Using the include element is important as otherwise AspectJ will look at every type loaded in support of your application (including all the Spring library classes and many more besides). Normally you don't want to weave these types and don't want to pay the overhead of AspectJ attempting to match against them.

To get informational messages in your log file regarding the activity of the load-time weaver, add the following options to the weaver element:

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN"    "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
   <weaver
         options="-showWeaveInfo
```

```
            -XmessageHandlerClass:org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
        <include within="com.xyz.myapp..*"/>
   </weaver>
</aspectj>
```

Finally, to control exactly which aspects are used, you can use the `aspects` element. By default all defined aspects are used for weaving (`spring-aspects.jar` contains a META-INF/aop.xml file that defines the configuration and transaction aspects). If you were using `spring-aspects.jar`, but only want the configuration support and not the transaction support you could specify this as follows:

```
<!DOCTYPE aspectj PUBLIC
  "-//AspectJ//DTD//EN"    "http://www.eclipse.org/aspectj/dtd/aspectj.dtd">

<aspectj>
  <weaver
    options="-showWeaveInfo -XmessageHandlerClass:org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
    <include within="com.xyz.myapp..*"/>
  </weaver>
  <aspects>
    <include within="org.springframework.beans.factory.aspectj.AnnotationBeanConfigurerAspect"/>
  </aspects>
</aspectj>
```

On the Java 5 platform, load-time weaving is enabled by specifying the following VM argument when launching the Java virtual machine:

```
-javaagent:<path-to-ajlibs>/aspectjweaver.jar
```

6.9. Further Resources

More information on AspectJ can be found at the AspectJ home page.

The book *Eclipse AspectJ* by Adrian Colyer et. al. (Addison-Wesley, 2005) provides a comprehensive introduction and reference for the AspectJ language.

The excellent *AspectJ in Action* by Ramnivas Laddad (Manning, 2003) comes highly recommended as an introduction to AOP; the focus of the book is on AspectJ, but a lot of general AOP themes are explored in some depth.

Chapter 7. Spring AOP APIs

7.1. Introduction

The previous chapter described the Spring 2.0 support for AOP using @AspectJ and schema-based aspect definitions. In this chapter we discuss the lower-level Spring AOP APIs and the AOP support used in Spring 1.2 applications. For new applications, we recommend the use of the Spring 2.0 AOP support described in the previous chapter, but when working with existing applications, or when reading books and articles, you may come across Spring 1.2 style examples. Spring 2.0 is fully backwards compatible with Spring 1.2 and everything described in this chapter is fully supported in Spring 2.0.

7.2. Pointcut API in Spring

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

7.2.1. 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.

Tip

If possible, try to make pointcuts static, allowing the AOP framework to cache the results of pointcut evaluation when an AOP proxy is created.

7.2.2. 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. However, using AspectJ pointcut expressions is usually a simpler approach.

7.2.3. AspectJ expression pointcuts

Since 2.0, the most important type of pointcut used by Spring is `org.springframework.aop.aspectj.AspectJExpressionPointcut`. This is a pointcut that uses an AspectJ supplied library to parse an AspectJ pointcut expression string.

See the previous chapter for a discussion of supported AspectJ pointcut primitives.

7.2.4. 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.

7.2.4.1. 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.

7.2.4.1.1. Regular expression pointcuts

One obvious way to specify static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible. `org.springframework.aop.support.Perl5RegexpMethodPointcut` is a generic regular expression pointcut, using Perl 5 regular expression syntax. The `Perl5RegexpMethodPointcut` class depends on Jakarta ORO for regular expression matching. Spring also provides the `JdkRegexpMethodPointcut` class that uses the regular expression support in JDK 1.4+.

Using the `Perl5RegexpMethodPointcut` 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.Perl5RegexpMethodPointcut">
    <property name="patterns">
        <list>
            <value>.*set.*</value>
            <value>.*absquatulate</value>
        </list>
    </property>
</bean>
```

Spring provides a convenience class, `RegexpMethodPointcutAdvisor`, that allows us to also reference an Advice (remember that an Advice can be an interceptor, before advice, throws advice etc.). Behind the scenes, Spring will use the `JdkRegexpMethodPointcut` on J2SE 1.4 or above, and will fall back to `Perl5RegexpMethodPointcut` on older VMs. The use of `Perl5RegexpMethodPointcut` can be forced by setting the `perl5` property to true. Using `RegexpMethodPointcutAdvisor` simplifies wiring, as the one bean encapsulates both pointcut and advice, as shown below:

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

RegexpMethodPointcutAdvisor can be used with any Advice type.

7.2.4.1.2. 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.

7.2.4.2. 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.

7.2.4.2.1. 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 a join point matched by another pointcut.) A control flow pointcut matches the current call stack. For example, it might fire if the join point 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.

7.2.5. 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 implementing just one abstract method (although it's possible to override other methods to customize behavior):

```
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.

7.2.6. Custom pointcuts

Because pointcuts in Spring AOP are Java classes, rather than language features (as in AspectJ) it's possible to declare custom pointcuts, whether static or dynamic. Custom pointcuts in Spring can be arbitrarily complex. However, using the AspectJ pointcut expression language is recommended if possible.

Note

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."

7.3. Advice API in Spring

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

7.3.1. Advice lifecycles

Each advice is a Spring bean. An advice instance 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.

7.3.2. 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.

7.3.2.1. 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 join point; the AOP proxy; and the arguments to the method. The `invoke()` method should return the invocation's result: the return value of the join point.

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 join point. 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!

Note

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.

7.3.2.2. 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 return type is `void`. Before advice can insert custom behavior before the join point 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 method invocations:

```
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;
    }
}
```

Tip

Before advice can be used with any pointcut.

7.3.2.3. Throws advice

Throws advice is invoked after the return of the join point if the join point 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 in the form of:

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

Only the last argument is required. The method signatures may have either one or four arguments, depending on whether the advice method is interested in the method and arguments. The following classes are examples of throws advice.

The advice below is 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 class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

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

Tip

Throws advice can be used with any pointcut.

7.3.2.4. 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.

Tip

After returning advice can be used with any pointcut.

7.3.2.5. 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 `IntroductionAdvisor`, which has the following methods:

```
public interface IntroductionAdvisor extends Advisor, IntroductionInfo {

        ClassFilter getClassFilter();

        void validateInterfaces() throws IllegalArgumentException;
}

public interface IntroductionInfo {

        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.
The `validateInterfaces()` method is used internally to see whether or not the introduced interfaces can be implemented by the configured `IntroductionInterceptor` .

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 `suppressInterface(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 join point - 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.

7.4. Advisor API in Spring

In Spring, an Advisor is an aspect that contains just a single advice object associated with a pointcut expression.

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 interceptor chain.

7.5. 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.)

Note

The Spring 2.0 AOP support also uses factory beans under the covers.

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.

7.5.1. Basics

The `ProxyFactoryBean`, like other Spring `FactoryBean` implementations, introduces a level of indirection. 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 another 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.

7.5.2. JavaBean properties

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

- Specify the target you want to proxy.
- Specify whether to use CGLIB (see below and also the section entitled Section 7.5.3, “JDK- and CGLIB-based proxies”).

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

- `proxyTargetClass`: `true` if the target class is to be proxied, rather than the target class' interfaces. If this property value is set to `true`, then CGLIB proxies will be created (but see also below the section entitled Section 7.5.3, “JDK- and CGLIB-based proxies”).
- `optimize`: controls whether or not aggressive optimizations are applied to proxies *created via CGLIB*. One should not blithely use this setting unless one fully understands how the relevant AOP proxy handles optimization. This is currently used only for CGLIB proxies; it has no effect with JDK dynamic proxies.
- `frozen`: if a proxy configuration is `frozen`, then changes to the configuration are no longer allowed. This is useful both as a slight optimization and for those cases when you don't want callers to be able to manipulate the proxy (via the `Advised` interface) after the proxy has been created. The default value of this property is `false`, so changes such as adding additional advice are allowed.
- `exposeProxy`: determines whether or not the current proxy should be exposed in a `ThreadLocal` so that it can be accessed by the target. If a target needs to obtain the proxy and the `exposeProxy` property is set to `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 is intended to allow the addition of new proxy types in Spring 1.1.

Other properties specific to `ProxyFactoryBean` include:

- `proxyInterfaces`: array of String interface names. If this isn't supplied, a CGLIB proxy for the target class will be used (but see also below the section entitled Section 7.5.3, “JDK- and CGLIB-based proxies”).
- `interceptorNames`: String array of `Advisor`, interceptor or other advice names to apply. Ordering is significant, on a first come-first served basis. That is to say that the first interceptor in the list will be the first to be able to intercept the invocation.

The names are bean names in the current factory, including bean names from ancestor factories. You can't mention bean references here since doing so would result in the `ProxyFactoryBean` ignoring the singleton setting of the advice.

You can append an interceptor name with an asterisk (`*`). This will result in the application of all advisor beans with names starting with the part before the asterisk to be applied. An example of using this feature can be found in Section 7.5.6, “Using 'global' advisors”.

- 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. The 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`.

7.5.3. JDK- and CGLIB-based proxies

This section serves as the definitive documentation on how the `ProxyFactoryBean` chooses to create one of either a JDK- and CGLIB-based proxy for a particular target object (that is to be proxied).

Note

The behavior of the `ProxyFactoryBean` with regard to creating JDK- or CGLIB-based proxies changed between versions 1.2.x and 2.0 of Spring. The `ProxyFactoryBean` now exhibits similar semantics with regard to auto-detecting interfaces as those of the `TransactionProxyFactoryBean` class.

If the class of a target object that is to be proxied (hereafter simply referred to as the target class) doesn't implement any interfaces, then a CGLIB-based proxy will be created. This is the easiest scenario, because JDK proxies are interface based, and no interfaces means JDK proxying isn't even possible. One simply plugs in the target bean, and specifies the list of interceptors via the `interceptorNames` property. Note that a CGLIB-based proxy will be created even if the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `false`. (Obviously this makes no sense, and is best removed from the bean definition because it is at best redundant, and at worst confusing.)

If the target class implements one (or more) interfaces, then the type of proxy that is created depends on the configuration of the `ProxyFactoryBean`.

If the `proxyTargetClass` property of the `ProxyFactoryBean` has been set to `true`, then a CGLIB-based proxy will be created. This makes sense, and is in keeping with the principle of least surprise. Even if the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, the fact that the `proxyTargetClass` property is set to `true` *will* cause CGLIB-based proxying to be in effect.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has been set to one or more fully qualified interface names, then a JDK-based proxy will be created. The created proxy will implement all of the interfaces that were specified in the `proxyInterfaces` property; if the target class happens to implement a whole lot more interfaces than those specified in the `proxyInterfaces` property, that is all well and good but those additional interfaces will not be implemented by the returned proxy.

If the `proxyInterfaces` property of the `ProxyFactoryBean` has *not* been set, but the target class *does implement one (or more)* interfaces, then the `ProxyFactoryBean` will auto-detect the fact that the target class does actually implement at least one interface, and a JDK-based proxy will be created. The interfaces that are actually

proxied will be *all* of the interfaces that the target class implements; in effect, this is the same as simply supplying a list of each and every interface that the target class implements to the `proxyInterfaces` property. However, it is significantly less work, and less prone to typos.

7.5.4. 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.DebugInterceptor">
</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.

Note

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, an 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).

It's possible to conceal the distinction between target and proxy using an anonymous *inner bean*, as follows. Only the `ProxyFactoryBean` definition is different; the advice is included only for completeness:

```
<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.DebugInterceptor"/>

<bean id="person" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces"><value>com.mycompany.Person</value></property>
  <!-- Use inner bean, not local reference to target -->
  <property name="target">
    <bean class="com.mycompany.PersonImpl">
      <property name="name"><value>Tony</value></property>
      <property name="age"><value>51</value></property>
    </bean>
  </property>
  <property name="interceptorNames">
    <list>
      <value>myAdvisor</value>
      <value>debugInterceptor</value>
    </list>
  </property>
</bean>
```

This has the advantage that there's only one object of type `Person`: useful if we want to prevent users of the application context from obtaining a reference to the un-advised object, or need to avoid any ambiguity with Spring IoC *autowiring*. There's also arguably an advantage in that the ProxyFactoryBean definition is self-contained. However, there are times when being able to obtain the un-advised target from the factory might actually be an *advantage*: for example, in certain test scenarios.

7.5.5. 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.

7.5.6. Using 'global' advisors

By appending an asterisk to an interceptor name, all advisors with bean names matching the part before the asterisk, will be added to the advisor chain. This can come in handy if you need to add a standard set of 'global' advisors:

```
<bean id="proxy" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="service"/>
  <property name="interceptorNames">
    <list>
      <value>global*</value>
    </list>
  </property>
</bean>

<bean id="global_debug" class="org.springframework.aop.interceptor.DebugInterceptor"/>
<bean id="global_performance" class="org.springframework.aop.interceptor.PerformanceMonitorInterceptor"/>
```

7.6. Concise proxy definitions

Especially when defining transactional proxies, you may end up with many similar proxy definitions. The use of parent and child bean definitions, along with inner bean definitions, can result in much cleaner and more concise proxy definitions.

First a parent, *template*, bean definition is created for the proxy:

```
<bean id="txProxyTemplate" abstract="true"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributes">
    <props>
      <prop key="*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

This will never be instantiated itself, so may actually be incomplete. Then each proxy which needs to be created is just a child bean definition, which wraps the target of the proxy as an inner bean definition, since the target will never be used on its own anyway.

```
<bean id="myService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MyServiceImpl">
    </bean>
  </property>
</bean>
```

It is of course possible to override properties from the parent template, such as in this case, the transaction

propagation settings:

```
<bean id="mySpecialService" parent="txProxyTemplate">
  <property name="target">
    <bean class="org.springframework.samples.MySpecialServiceImpl">
    </bean>
  </property>
  <property name="transactionAttributes">
    <props>
      <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="find*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="load*">PROPAGATION_REQUIRED,readOnly</prop>
      <prop key="store*">PROPAGATION_REQUIRED</prop>
    </props>
  </property>
</bean>
```

Note that in the example above, we have explicitly marked the parent bean definition as *abstract* by using the *abstract* attribute, as described previously, so that it may not actually ever be instantiated. Application contexts (but not simple bean factories) will by default pre-instantiate all singletons. It is therefore important (at least for singleton beans) that if you have a (parent) bean definition which you intend to use only as a template, and this definition specifies a class, you must make sure to set the *abstract* attribute to *true*, otherwise the application context will actually try to pre-instantiate it.

7.7. 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 construct an 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`) which allow you to add other advice types such as before and throws advice. AdvisedSupport is the superclass of both ProxyFactory and ProxyFactoryBean.

Tip

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.

7.8. 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, whichever other interfaces it implements. This interface includes the following methods:

```
Advisor[] getAdvisors();

void addAdvice(Advice advice) throws AopConfigException;

void addAdvice(int pos, Advice advice)
        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 the advisor holding pointcut and advice will be the generic `DefaultPointcutAdvisor`, which can be used with any advice or pointcut (but not for introductions).

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.)

A simple example of casting an AOP proxy to the `Advised` interface and examining and manipulating its advice:

```
Advised advised = (Advised) myObject;
Advisor[] advisors = advised.getAdvisors();
int oldAdvisorCount = advisors.length;
System.out.println(oldAdvisorCount + " advisors");

// Add an advice like an interceptor without a pointcut
// Will match all proxied methods
// Can use for interceptors, before, after returning or throws advice
advised.addAdvice(new DebugInterceptor());

// Add selective advice using a pointcut
advised.addAdvisor(new DefaultPointcutAdvisor(mySpecialPointcut, myAdvice));

assertEquals("Added two advisors",
     oldAdvisorCount + 2, advised.getAdvisors().length);
```

Note

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.

7.9. 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 to configure 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.

7.9.1. Autoproxy bean definitions

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

7.9.1.1. BeanNameAutoProxyCreator

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

```
<bean 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 interceptors, to allow correct behavior 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, 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.

7.9.1.2. DefaultAdvisorAutoProxyCreator

A more general and extremely powerful auto proxy creator is `DefaultAdvisorAutoProxyCreator`. 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 a `DefaultAdvisorAutoProxyCreator` 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 definitions.

The `DefaultAdvisorAutoProxyCreator` 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. (The "inner bean" idiom shown earlier also offers this benefit.)

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

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

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

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

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

The `DefaultAdvisorAutoProxyCreator` 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 DefaultAdvisorAutoProxyCreator 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 TransactionAttributeSourceAdvisor used in the above example has a configurable order value; the default setting is unordered.

7.9.1.3. AbstractAdvisorAutoProxyCreator

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

7.9.2. Using metadata-driven auto-proxying

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 `DefaultAdvisorAutoProxyCreator`, 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 `DefaultAdvisorAutoProxyCreator`, 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 class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
      <property name="attributes" ref="attributes"/>
    </bean>
  </property>
</bean>

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

The `DefaultAdvisorAutoProxyCreator` bean definition (the name is not significant, hence it can even be omitted) will pick up all eligible pointcuts in the current application context. In this case, the "transactionAdvisor" bean definition, of type `TransactionAttributeSourceAdvisor`, will apply to classes or methods carrying a transaction attribute. The TransactionAttributeSourceAdvisor 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 fragment, 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 `/annotation` directory of the JPetStore sample application contains an analogous example for auto-proxying driven by JDK 1.5+ annotations. The following configuration enables automatic detection of Spring's `Transactional` annotation, leading to implicit proxies for beans containing that annotation:

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

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="transactionInterceptor"
    class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
  </property>
</bean>
```

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"/>
```

Tip

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"
    scope="prototype"/>

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

7.10. 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 join point. 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.

Tip

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.

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

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

```
</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.

7.10.2. 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 1.3, 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"
    scope="prototype">
  ... properties omitted
</bean>

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

<bean id="businessObject" class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="targetSource" ref="poolTargetSource"/>
  <property name="interceptorNames" value="myInterceptor"/>
</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="targetObject" ref="poolTargetSource"/>
  <property name="targetMethod" value="getPoolingConfigMixin"/>
</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());
```

Note

Pooling stateless service objects is not usually necessary. We don't believe it should be the default choice, as most stateless objects are naturally thread safe, 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.

7.10.3. 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" ref="businessObjectTarget"/>
</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.

7.10.4. `ThreadLocal` target sources

`ThreadLocal` target sources are useful if you need an object to be created for each incoming request (per thread that is). The concept of a `ThreadLocal` provide a JDK-wide facility to transparently store resource alongside a thread. Setting up a `ThreadLocalTargetSource` is pretty much the same as was explained for the other types of target source:

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

Note

ThreadLocals come with serious issues (potentially resulting in memory leaks) when incorrectly using them in a multi-threaded and multi-classloader environments. One should always consider wrapping a threadlocal in some other class and never directly use the `ThreadLocal` itself (except of

course in the wrapper class). Also, one should always remember to correctly set and unset (where the latter simply involved a call to `ThreadLocal.set(null)`) the resource local to the thread. Unsetting should be done in any case since not unsetting it might result in problematic behavior. Spring's ThreadLocal support does this for you and should always be considered in favor of using ThreadLocals without other proper handling code.

7.11. 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 the out-of-the-box interception around advice, before, throws advice and after returning advice.

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.

7.12. Further resources

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.

Chapter 8. Testing

8.1. Introduction

The Spring team considers developer testing to be an absolutely integral part of enterprise software development. A thorough treatment of testing in the enterprise is beyond the scope of this chapter; rather, the focus here is on the value add that the adoption of the IoC principle can bring to unit testing; and on the benefits that the Spring Framework provides in integration testing.

8.2. Unit testing

One of the main benefits of Dependency Injection is that your code should really depend far less on the container than in traditional J2EE development. The POJOs that comprise your application should be testable in JUnit tests, with objects simply instantiated using the `new` operator, *without Spring or any other container*. You can use mock objects (in conjunction with many other valuable testing techniques) to test your code in isolation. If you follow the architecture recommendations around Spring you will find that the resulting clean layering and componentization of your codebase will naturally faciliate *easier* unit testing. For example, you will be able to test service layer objects by stubbing or mocking DAO interfaces, without any need to access persistent data while running unit tests.

True unit tests typically will run extremely quickly, as there is no runtime infrastructure to set up, whether application server, database, ORM tool, or whatever. Thus emphasizing true unit tests as part of your development methodology will boost your productivity. The upshot of this is that you do not need this section of the testing chapter to help you write effective *unit* tests for your IoC-based applications.

8.3. Integration testing

However, it is also important to be able to perform some integration testing without requiring deployment to your application server or connecting to other enterprise infrastructure. This will enable you to test things such as:

- The correct wiring of your Spring IoC container contexts.
- Data access using JDBC or an ORM tool. This would include such things such as the correctness of SQL statements / or Hibernate XML mapping files.

The Spring Framework provides first class support for integration testing in the form of the classes that are packaged in the `spring-mock.jar` library. *Please note that these test classes are JUnit-specific.*

The `org.springframework.test` package provides valuable JUnit `TestCase` superclasses for integration testing using a Spring container, while at the same time not being reliant on an application server or other deployed environment. They will be slower to run than unit tests, but much faster to run than the equivalent Cactus tests or remote tests relying on deployment to an application server.

These superclasses provide the following functionality:

- Spring IoC container caching between test case execution.

- The pretty-much-transparent Dependency Injection of test fixture instances (this is nice).
- Transaction management appropriate to integration testing (this is even nicer).
- A number of Spring-specific inherited instance variables that are really useful when integration testing.

8.3.1. Context management and caching

The `org.springframework.test` package provides support for consistent loading of Spring contexts, and caching of loaded contexts. Support for the caching of loaded contexts is important, because if you are working on a large project, startup time may become an issue - not because of the overhead of Spring itself, but because the objects instantiated by the Spring container will themselves take time to instantiate. For example, a project with 50-100 Hibernate mapping files might take 10-20 seconds to load the mapping files, and incurring that cost before running every single test case in every single test fixture will lead to slower overall test runs that could reduce productivity.

To address this issue, the `AbstractDependencyInjectionSpringContextTests` has an `protected` method that subclasses must implement to provide the location of context definition files:

```
protected String[] getConfigLocations();
```

Implementations of this method must provide an array containing the resource locations of XML configuration metadata - typically on the classpath - used to configure the application. This will be the same, or nearly the same, as the list of configuration locations specified in `web.xml` or other deployment configuration.

By default, once loaded, the configuration fileset will be reused for each test case. Thus the setup cost will be incurred only once (per test fixture), and subsequent test execution will be much faster. In the unlikely case that a test may 'dirty' the config location, requiring reloading - for example, by changing a bean definition or the state of an application object - you can call the `setDirty()` method on `AbstractDependencyInjectionSpringContextTests` to cause the test fixture to reload the configurations and rebuild the application context before executing the next test case.

8.3.2. Dependency Injection of test fixtures

When `AbstractDependencyInjectionSpringContextTests` (and subclasses) load your application context, they can optionally configure instances of your test classes by Setter Injection. All you need to do is to define instance variables and the corresponding setters. `AbstractDependencyInjectionSpringContextTests` will automatically locate the corresponding object in the set of configuration files specified in the `getConfigLocations()` method.

Consider the scenario where we have a class, `HibernateTitleDao`, that performs data access logic for say, the `Title` domain object. We want to write integration tests that test all of the following areas:

- The Spring configuration; basically, is everything related to the configuration of the `HibernateTitleDao` bean correct and present?
- The Hibernate mapping file configuration; is everything mapped correctly and are the correct lazy-loading settings in place?
- The logic of the `HibernateTitleDao`; does the configured instance of this class perform as anticipated?

Let's look at the test class itself (we will look at the configuration immediately afterwards).

```
public final class HibernateTitleDaoTests extends AbstractDependencyInjectionSpringContextTests  {

    // this instance will be (automatically) dependency injected
    private HibernateTitleDao titleDao;

    // a setter method to enable DI of the 'titleDao' instance variable
    public void setTitleDao(HibernateTitleDao titleDao) {
        this.titleDao = titleDao;
    }

    public void testLoadTitle() throws Exception {
        Title title = this.titleDao.loadTitle(new Long(10));
        assertNotNull(title);
    }

    // specifies the Spring configuration to load for this test fixture
    protected String[] getConfigLocations() {
        return new String[] { "classpath:com/foo/daos.xml" };
    }

}
```

The file referenced by the `getConfigLocations()` method (`'classpath:com/foo/daos.xml'`) looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC  "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <!-- this bean will be injected into the HibernateTitleDaoTests class -->
    <bean id="titleDao" class="com.foo.dao.hibernate.HibernateTitleDao">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <bean id="sessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <!-- dependencies elided for clarity -->
    </bean>

</beans>
```

The `AbstractDependencyInjectionSpringContextTests` classes uses *autowire by type*. Thus if you have multiple bean definitions of the same type, you cannot rely on this approach for those particular beans. In that case, you can use the inherited `applicationContext` instance variable, and explicit lookup using (for example) an explicit call to `applicationContext.getBean("titleDao")`.

If you don't want dependency injection applied to your test cases, simply don't declare any setters. Alternatively, you can extend the `AbstractSpringContextTests` - the root of the class hierarchy in the `org.springframework.test` package. It merely contains convenience methods to load Spring contexts, and performs no Dependency Injection of the test fixture.

8.3.2.1. Field level injection

If, for whatever reason, you don't fancy having setter methods in your test fixtures, Spring can (in this one case) inject dependencies into `protected` fields. Find below a reworking of the previous example to use field level injection (the Spring XML configuration does not need to change, merely the test fixture).

```
public final class HibernateTitleDaoTests extends AbstractDependencyInjectionSpringContextTests  {

    public HibernateTitleDaoTests() {
            // switch on field level injection
        setPopulateProtectedVariables(true);
    }

    // this instance will be (automatically) dependency injected
    protected HibernateTitleDao titleDao;

    public void testLoadTitle() throws Exception {
```

```
        Title title = this.titleDao.loadTitle(new Long(10));
        assertNotNull(title);
    }

    // specifies the Spring configuration to load for this test fixture
    protected String[] getConfigLocations() {
        return new String[] { "classpath:com/foo/daos.xml" };
    }

}
```

In the case of field injection, there is no autowiring going on: the name of your `protected` instances variable(s) are used as the lookup bean name in the configured Spring container.

8.3.3. Transaction management

One common issue in tests that access a real database is their affect on the state of the persistence store. Even when you're using a development database, changes to the state may affect future tests. Also, many operations - such as inserting to or modifying persistent data - cannot be done (or verified) outside a transaction.

The `org.springframework.test.AbstractTransactionalDataSourceSpringContextTests` superclass (and subclasses) exist to meet this need. By default, they create and roll back a transaction for each test. You simply write code that can assume the existence of a transaction. If you call transactionally proxied objects in your tests, they will behave correctly, according to their transactional semantics.

`AbstractTransactionalSpringContextTests` depends on a `PlatformTransactionManager` bean being defined in the application context. The name doesn't matter, due to the use of autowire by type.

Typically you will extend the subclass, `AbstractTransactionalDataSourceSpringContextTests`. This also requires that a `DataSource` bean definition - again, with any name - be present in the configurations. It creates a `JdbcTemplate` instance variable that is useful for convenient querying, and provides handy methods to delete the contents of selected tables (remember that the transaction will roll back by default, so this is safe to do).

If you want a transaction to commit - unusual, but occasionally useful when you want a particular test to populate the database - you can call the `setComplete()` method inherited from `AbstractTransactionalSpringContextTests`. This will cause the transaction to commit instead of roll back.

There is also convenient ability to end a transaction before the test case ends, through calling the `endTransaction()` method. This will roll back the transaction by default, and commit it only if `setComplete()` had previously been called. This functionality is useful if you want to test the behavior of 'disconnected' data objects, such as Hibernate-mapped objects that will be used in a web or remoting tier outside a transaction. Often, lazy loading errors are discovered only through UI testing; if you call `endTransaction()` you can ensure correct operation of the UI through your JUnit test suite.

8.3.4. Convenience variables

When you extend the `AbstractTransactionalDataSourceSpringContextTests` class you will have access to the following `protected` instance variables:

- `applicationContext` (a `ConfigurableApplicationContext`): inherited from the `AbstractDependencyInjectionSpringContextTests` superclass. Use this to perfom explicit bean lookup, or test the state of the context as a whole.
- `jdbcTemplate`: inherited from `AbstractTransactionalDataSourceSpringContextTests`. Useful for querying to confirm state. For example, you might query before and after testing application code that creates

an object and persists it using an ORM tool, to verify that the data appears in the database. (Spring will ensure that the query runs in the scope of the same transaction.) You will need to tell your ORM tool to 'flush' its changes for this to work correctly, for example using the `flush()` method on Hibernate's `Session` interface.

Often you will provide an application-wide superclass for integration tests that provides further useful instance variables used in many tests.

8.3.5. Java 5+ specific support

If you are developing against Java 5 or greater, there are some additional annotations and support classes that you can use in your testing. The `AbstractAnnotationAwareTransactionalTests` class extends the `AbstractTransactionalDataSourceSpringContextTests` makes the text fixtures that you write that inherit from it aware of a number of (Spring-specific) annotations.

8.3.5.1. Annotations

The Spring Framework provides a number of annotations to help when writing integration tests. Please note that these annotations *must* be used in conjunction with the aforementioned `AbstractAnnotationAwareTransactionalTests` in order for the presence of these annotations to have any effect.

- `@DirtiesContext`.

  The presence of this annotation on a text method indicates that the underlying Spring container is 'dirtied' during the execution of of the test method, and thus must be rebuilt after the test method finishes execution (regardless of whether the test passed or not). Has the same effect as a regular `setDirty()` invocation.

```
@DirtiesContext
public void testProcess() {
        // some logic that results in the Spring container being dirtied
}
```

- `@ExpectedException`.

  Indicates that the annotated test method is expected to throw an exception during execution. The type of the expected exception is provided in the annotation, and if an an instance of the exception is thrown during the test method execution then the test passes. Likewise if an instance of the exception is *not* thrown during the test method execution then the test fails.

```
@ExpectedException(SomeBusinessException.class)
public void testProcessRainyDayScenario() {
        // some logic that results in an Exception being thrown
}
```

- `@NotTransactional`.

  Simply indicates that the annotated test method must *not* execute in a transactional context.

```
@NotTransactional
public void testProcess() {
        // ...
}
```

- `@Repeat`

  Indicates that the annotated test method must be executed repeatedly. The number of times that the test method is to be executed is specified in the annotation.

```
@Repeat(10)
public void testProcessRepeatedly() {
        // ...
}
```

8.3.6. PetClinic example

The PetClinic sample application included with the Spring distribution illustrates the use of these test superclasses. Most test functionality is included in the `AbstractClinicTests`, for which a partial listing is shown below:

```
public abstract class AbstractClinicTests
               extends AbstractTransactionalDataSourceSpringContextTests {

   protected Clinic clinic;

   public void setClinic(Clinic clinic) {
      this.clinic = clinic;
   }

   public void testGetVets() {
      Collection vets = this.clinic.getVets();
      assertEquals('JDBC query must show the same number of vets',
         jdbcTemplate.queryForInt('SELECT COUNT(0) FROM VETS'),
         vets.size());
      Vet v1 = (Vet) EntityUtils.getById(vets, Vet.class, 2);
      assertEquals('Leary', v1.getLastName());
      assertEquals(1, v1.getNrOfSpecialties());
      assertEquals('radiology', ((Specialty) v1.getSpecialties().get(0)).getName());
      Vet v2 = (Vet) EntityUtils.getById(vets, Vet.class, 3);
      assertEquals('Douglas', v2.getLastName());
      assertEquals(2, v2.getNrOfSpecialties());
      assertEquals('dentistry', ((Specialty) v2.getSpecialties().get(0)).getName());
      assertEquals('surgery', ((Specialty) v2.getSpecialties().get(1)).getName());
}
```

Notes:

- This test case extends the `AbstractTransactionalDataSourceSpringContextTests` class, from which it inherits Dependency Injection and transactional behavior.
- The `clinic` instance variable - the application object being tested - is set by Dependency Injection through the `setClinic(..)` method.
- The `testGetVets()` method illustrates how the inherited `JdbcTemplate` variable can be used to verify correct behavior of the application code being tested. This allows for stronger tests, and lessens dependency on the exact test data. For example, you can add additional rows in the database without breaking tests.
- Like many integration tests using a database, most of the tests in `AbstractClinicTests` depend on a minimum amount of data already in the database before the test cases run. You might, however, choose to populate the database in your test cases also - again, within the one transaction.

The PetClinic application supports four data access technologies - JDBC, Hibernate, TopLink, and JPA. Thus the `AbstractClinicTests` class does not itself specify the context locations - this is deferred to subclasses, that implement the necessary protected abstract method from `AbstractDependencyInjectionSpringContextTests`.

For example, the Hibernate implementation of the PetClinic tests contains the following implementation:

```
public final class HibernateClinicTests extends AbstractClinicTests {

   protected String[] getConfigLocations() {
      return new String[] {
         "/org/springframework/samples/petclinic/hibernate/applicationContext-hibernate.xml"
      };
   }
}
```

As the PetClinic is a very simple application, there is only one Spring configuration file. Of course, more complex applications will typically break their Spring configuration across multiple files. Instead of being defined in a leaf class, config locations will often be specified in a common base class for all application-specific integration tests. This may also add useful instance variables - populated by Dependency Injection, naturally - such as a `HibernateTemplate`, in the case of an application using Hibernate.

As far as possible, you should have exactly the same Spring configuration files in your integration tests as in the deployed environment. One likely point of difference concerns database connection pooling and transaction infrastructure. If you are deploying to a full-blown application server, you will probably use its connection pool (available through JNDI) and JTA implementation. Thus in production you will use a `JndiObjectFactoryBean` for the `DataSource`, and `JtaTransactionManager`. JNDI and JTA will not be available in out-of-container integration tests, so you should use a combination like the Commons DBCP `BasicDataSource` and `DataSourceTransactionManager` or `HibernateTransactionManager` for them. You can factor out this variant behavior into a single XML file, having the choice between application server and 'local' configuration separated from all other configuration, which will not vary between the test and production environments.

8.4. Further Resources

This section contains links to further resources about testing in general.

- The JUnit homepage. The Spring Framework's unit test suite is written using JUnit as the testing framework.
- The EasyMock homepage. The Spring Framework uses EasyMock extensively in it's test suite.
- The JMock homepage.
- The DbUnit homepage.
- The Grinder homepage (load testing framework).

Part II. Middle Tier Data Access

This part of the reference documentation is concerned with the middle tier, and specifically the data access responsibilities of said tier.

Spring's comprehensive transaction management support is covered in some detail, followed by thorough coverage of the various middle tier data access frameworks and technologies that the Spring Framework integrates with.

- Chapter 9, *Transaction management*
- Chapter 10, *DAO support*
- Chapter 11, *Data access using JDBC*
- Chapter 12, *Object Relational Mapping (ORM) data access*

Chapter 9. Transaction management

9.1. Introduction

One of the most compelling reasons to use the Spring Framework is the comprehensive transaction support. The Spring Framework provides a consistent abstraction for transaction management that delivers the following benefits:

- Provides a consistent programming model across different transaction APIs such as JTA, JDBC, Hibernate, JPA, and JDO.
- Supports declarative transaction management.
- Provides a simpler API for programmatic transaction management than a number of complex transaction APIs such as JTA.
- Integrates very well with Spring's various data access abstractions.

This chapter is divided up into a number of sections, each detailing one of the value-adds or technologies of the Spring Framework's transaction support. The chapter closes up with some discussion of best practices surrounding transaction management (for example, choosing between declarative and programmatic transaction management).

- The first section, entitled Motivations, describes *why* one would want to use the Spring Framework's transaction abstraction as opposed to EJB CMT or driving transactions via a proprietary API such as Hibernate.
- The second section, entitled Key abstractions outlines the core classes in the Spring Framework's transaction support, as well as how to configure and obtain `DataSource` instances from a variety of sources.
- The third section, entitled Declarative transaction management, covers the Spring Framework's support for declarative transaction management.
- The fourth section, entitled Programmatic transaction management, covers the Spring Framework's support for programmatic (that is, explicitly coded) transaction management.

9.2. Motivations

Is an application server needed for transaction management?

The Spring Framework's transaction management support 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 the Spring Framework's declarative transactions offer more power and a much more productive programming model than EJB CMT.

Typically you need an application server's JTA capability only if you need to enlist multiple transactional

resources, and for many applications being able to handle transactions across multiple resources isn't a requirement. For example, many high-end applications use a single, highly scalable database (such as Oracle 9i RAC). Standalone transaction managers such as Atomikos Transactions and JOTM are other options. (Of course you may need other application server capabilities such as JMS and JCA.)

The most important point is that with the Spring Framework *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 code 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 the Spring Framework, only configuration needs to change so that your code doesn't have to.

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

Global Transactions. Global transactions have a significant downside, in that code needs to use JTA, and JTA is a cumbersome API to use (partly due to its exception model). Furthermore, a JTA `UserTransaction` normally needs to be sourced 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. Previously, the preferred way to use global transactions was via EJB *CMT* (*Container Managed Transaction*): CMT is 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 of the need (although not entirely) to write Java code to control transactions. The significant downside is that CMT is tied to JTA and an application server environment. Also, it is only available if one chooses 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, especially in the face of compelling alternatives for declarative transaction management.

Local Transactions. Local transactions may be easier to use, but have significant disadvantages: they cannot work across multiple transactional resources. For example, code that manages transactions using a JDBC connection cannot run within a global JTA transaction. Another downside is that local transactions tend to be invasive to the programming model.

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. The Spring Framework provides both declarative and programmatic transaction management. Declarative transaction management is preferred by most users, and is recommended in most cases.

With programmatic transaction management, developers work with the Spring Framework 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 the Spring Framework's transaction API (or indeed on any other transaction API).

9.3. Key abstractions

The key to the Spring transaction abstraction is the notion of a *transaction strategy*. A transaction strategy is defined by 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 the Spring Framework's philosophy, `PlatformTransactionManager` is an *interface*, and can thus be easily mocked or stubbed as necessary. Nor is it tied to a lookup strategy such as JNDI: `PlatformTransactionManager` implementations are defined like any other object (or bean) in the Spring Framework's IoC container. This benefit alone makes it a worthwhile abstraction even when working with JTA: transactional code can be tested much more easily than if it used JTA directly.

Again in keeping with Spring's philosophy, the `TransactionException` that can be thrown by any of the `PlatformTransactionManager` interface's methods is *unchecked* (i.e. it extends the `java.lang.RuntimeException` class). Transaction infrastructure failures are almost invariably fatal. In rare cases where application code can actually recover from a transaction failure, the application developer can still choose to catch and handle `TransactionException`. The salient point is that developers are not *forced* to do so.

The `getTransaction(..)` method returns a `TransactionStatus` object, depending on a `TransactionDefinition` parameter. The returned `TransactionStatus` might represent a new or existing transaction (if there were a matching transaction in the current call stack - with the implication being that (as with J2EE transaction contexts) a `TransactionStatus` is associated with a **thread** of execution).

The `TransactionDefinition` interface specifies:

- **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?
- **Propagation**: normally all code executed within a transaction scope will run in that transaction. However, there are several options specifying behavior if a transactional method is executed when a transaction context already exists: for example, simply continue running in the existing transaction (the common case); or suspending the existing transaction and creating a new transaction. *Spring offers all of the transaction propagation options familiar from EJB CMT*.
- **Timeout**: how long this transaction may run before timing out (and 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 transactional concepts. If necessary, please refer to a resource discussing transaction isolation levels and other core transaction concepts because understanding such core concepts is essential to using the Spring Framework or indeed 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();
}
```

Regardless of whether you opt for declarative or programmatic transaction management in Spring, defining the correct `PlatformTransactionManager` implementation is absolutely essential. In good Spring fashion, this important definition typically is made using via Dependency Injection.

`PlatformTransactionManager` implementations normally require knowledge of the environment in which they work: JDBC, JTA, Hibernate, etc The following examples from the `dataAccessContext-local.xml` file from Spring's **jPetStore** sample application show how a local `PlatformTransactionManager` implementation can be defined. (This will work with plain 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="${jdbc.driverClassName}" />
  <property name="url" value="${jdbc.url}" />
  <property name="username" value="${jdbc.username}" />
  <property name="password" value="${jdbc.password}" />
</bean>
```

The related `PlatformTransactionManager` bean definition will look like this:

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

If we use JTA in a J2EE container, as in the `'dataAccessContext-jta.xml'` file from the same sample application, we use a container `DataSource`, obtained via JNDI, in conjunction with Spring's `JtaTransactionManager`. 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 infrastructure.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.
       http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

  <jee:jndi-lookup id="dataSource" jndi-name="jdbc/jpetstore"/>

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

  <!-- other <bean/> definitions here -->

</beans>
```

Note

The above definition of the `'dataSource'` bean uses the `<jndi-lookup/>` tag from the `'jee'` namespace. For more information on schema-based configuration, see Appendix A, *XML Schema-based configuration*, and for more information on the `<jee/>` tags see the section entitled

Section A.2.3, “The `jee` schema”.

We can also use Hibernate local transactions easily, as shown in the following examples from the Spring Framework's **PetClinic** sample application. In this case, we need to define a Hibernate `LocalSessionFactoryBean`, which application code will use to obtain Hibernate `Session` instances.

The `DataSource` bean definition will be similar to the one shown previously (and thus is not shown). If the `DataSource` is managed by the JEE container it should be non-transactional as the Spring Framework, rather than the JEE container, will manage transactions.

The `'txManager'` bean in this case is of the `HibernateTransactionManager` type. In the same way as the `DataSourceTransactionManager` needs a reference to the `DataSource`, the `HibernateTransactionManager` needs a reference to the `SessionFactory`.

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

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

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

```
<bean id="txManager" 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 will not 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.

9.4. Resource synchronization with transactions

It should now be clear how different transaction managers are created, and how they are linked to related resources which need to be synchronized to transactions (i.e. `DataSourceTransactionManager` to a JDBC `DataSource`, `HibernateTransactionManager` to a Hibernate `SessionFactory`, etc). There remains the question however of how the application code, directly or indirectly using a persistence API (JDBC, Hibernate, JDO, etc), ensures that these resources are obtained and handled properly in terms of proper creation/reuse/cleanup and trigger (optionally) transaction synchronization via the relevant `PlatformTransactionManager`.

9.4.1. High-level approach

The preferred approach is to use Spring's highest level persistence integration APIs. These do not replace the native APIs, but internally handle resource creation/reuse, cleanup, optional transaction synchronization of the resources and exception mapping so that user data access code doesn't have to worry about these concerns at all, but can concentrate purely on non-boilerplate persistence logic. Generally, the same *template* approach is used for all persistence APIs, with examples including the `JdbcTemplate`, `HibernateTemplate`, and `JdoTemplate` classes (detailed in subsequent chapters of this reference documentation.

9.4.2. Low-level approach

At a lower level exist classes such as `DataSourceUtils` (for JDBC), `SessionFactoryUtils` (for Hibernate), `PersistenceManagerFactoryUtils` (for JDO), and so on. When it is preferable for application code to deal directly with the resource types of the native persistence APIs, these classes ensure that proper Spring Framework-managed instances are obtained, transactions are (optionally) synchronized, and exceptions which happen in the process are properly mapped to a consistent API.

For example, in the case of JDBC, instead of the traditional JDBC approach of calling the `getConnection()` method on the `DataSource`, you would instead use Spring's `org.springframework.jdbc.datasource.DataSourceUtils` class as follows:

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

If an existing transaction exists, and already has a connection synchronized (linked) to it, that instance will be returned. Otherwise, the method call will trigger the creation of a new connection, which will be (optionally) synchronized to any existing transaction, and made available for subsequent reuse in that same transaction. As mentioned, this has the added advantage that any `SQLException` will be wrapped in a Spring Framework `CannotGetJdbcConnectionException` - one of the Spring Framework'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.

It should be noted that this will also work fine without Spring transaction management (transaction synchronization is optional), 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 will generally prefer not 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.

9.4.3. `TransactionAwareDataSourceProxy`

At the very lowest level exists the `TransactionAwareDataSourceProxy` class. This is a proxy for a target `DataSource`, which wraps the target `DataSource` to add awareness of Spring-managed transactions. In this respect, it is similar to a transactional JNDI `DataSource` as provided by a J2EE server.

It should almost never be necessary or desirable to use this class, except when existing code exists which must be called and passed a standard JDBC `DataSource` interface implementation. In that case, it's possible to still have this code be usable, but participating in Spring managed transactions. It is preferable to write your new code using the higher level abstractions mentioned above.

9.5. Declarative transaction management

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

The Spring Framework's declarative transaction management is made possible with Spring AOP, although, as the transactional aspects code comes with the Spring Framework distribution and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with the Spring Framework's declarative transaction management. The basic approach is similar: it is possible to specify transaction behavior (or lack of it) down to individual method level. It is possible to make a `setRollbackOnly()` call within a transaction context if necessary. The differences are:

- Unlike EJB CMT, which is tied to JTA, the Spring Framework's declarative transaction management works in any environment. It can work with JDBC, JDO, Hibernate or other transactions under the covers, with configuration changes only.
- The Spring Framework enables declarative transaction management to be applied to any class, not merely special classes such as EJBs.
- The Spring Framework offers declarative *rollback rules*: a feature with no EJB equivalent, which we'll discuss below. Rollback can be controlled declaratively, not merely programmatically.
- The Spring Framework gives you an opportunity to customize transactional behavior, using AOP. For example, if you want to insert custom behavior 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()`.
- The Spring Framework 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, consider carefully before using such a feature. Normally, we do not want transactions to span remote calls.

Where is `TransactionProxyFactoryBean`?

Declarative transaction configuration in versions of Spring 2.0 and above differs considerably from previous versions of Spring. The main difference is that there is no longer any need to configure `TransactionProxyFactoryBean` beans.

The old, pre-Spring 2.0 configuration style is still 100% valid configuration; under the covers think of the new `<tx:tags/>` as simply defining `TransactionProxyFactoryBean` beans on your behalf.

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` must always result in rollback. 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 behavior is 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* (i.e. a checked exception other than `java.rmi.RemoteException`). While the Spring default behavior for declarative transaction management follows EJB convention (roll back is automatic only on unchecked exceptions), it is often useful to customize this.

9.5.1. Understanding the Spring Framework's declarative transaction implementation

The aim of this section is to dispel the mystique that is sometimes associated with the use of declarative transactions. It is all very well for this reference documentation simply to tell you to annotate your classes with the `@Transactional` annotation, add the line (`'<tx:annotation-driven/>'`) to your configuration, and then expect you to understand how it all works. This section will explain the inner workings of the Spring Framework's declarative transaction infrastructure to help you navigate your way back upstream to calmer waters in the event of transaction-related issues.

Tip

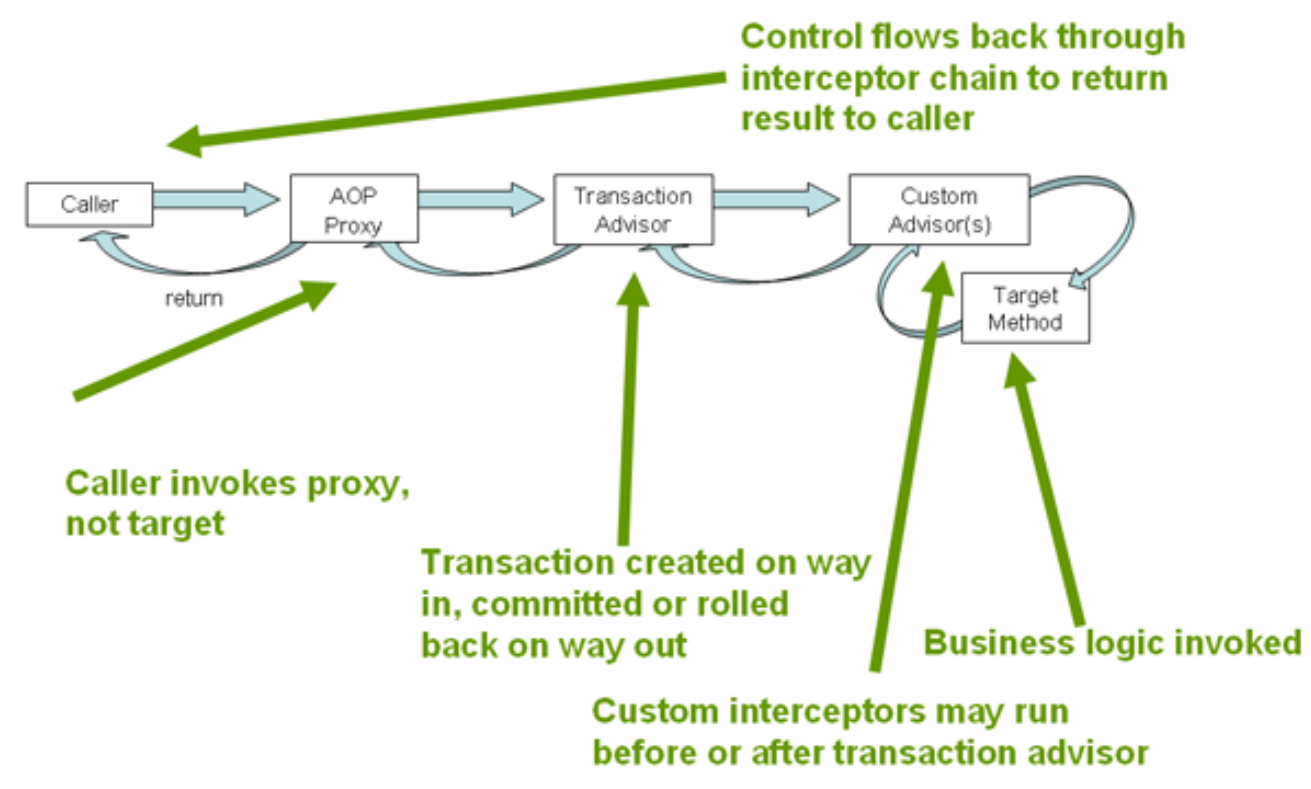
Looking at the Spring Framework source code is a good way to get a real understanding of the transaction support. We also suggest turning the logging level to `'DEBUG'` during development to better see what goes on under the hood.

The most important concepts to grasp with regard to the Spring Framework's declarative transaction support are that this support is enabled *via AOP proxies*, and that the transactional advice is driven by *metadata* (currently XML- or annotation-based). The combination of AOP with transactional metadata yields an AOP proxy that uses a `TransactionInterceptor` in conjunction with an appropriate `PlatformTransactionManager` implementation to drive transactions *around method invocations*.

Note

Although knowledge of Spring AOP is not required to use Spring's declarative transaction support, it can help. Spring AOP is thoroughly covered in the chapter entitled Chapter 6, *Aspect Oriented Programming with Spring*.

Conceptually, calling a method on a transactional proxy looks like this...



9.5.2. A first example

Consider the following interface, and its attendant implementation. (The intent is to convey the concepts, and using the rote `Foo` and `Bar` tropes means that you can concentrate on the transaction usage and not have to worry about the domain model.)

```
// the service interface that we want to make transactional

package x.y.service;

public interface FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);

}
```

```
// an implementation of the above interface

package x.y.service;

public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        throw new UnsupportedOperationException();
    }

    public Foo getFoo(String fooName, String barName) {
        throw new UnsupportedOperationException();
    }

    public void insertFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

    public void updateFoo(Foo foo) {
        throw new UnsupportedOperationException();
    }

}
```

(For the purposes of this example, the fact that the `DefaultFooService` class throws `UnsupportedOperationException` instances in the body of each implemented method is good; it will allow us to see transactions being created and then rolled back in response to the `UnsupportedOperationException` instance being thrown.)

Let's assume that the first two methods of the `FooService` interface (`getFoo(String)` and `getFoo(String, String)`) have to execute in the context of a transaction with read-only semantics, and that the other methods (`insertFoo(Foo)` and `updateFoo(Foo)`) have to execute in the context of a transaction with read-write semantics. Don't worry about taking the following configuration in all at once; everything will be explained in detail in the next few paragraphs.

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
```

```
      http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- the transactional advice (i.e. what 'happens'; see the <aop:advisor/> bean below) -->
  <tx:advice id="txAdvice" transaction-manager="txManager">
    <!-- the transactional semantics... -->
    <tx:attributes>
      <!-- all methods starting with 'get' are read-only -->
      <tx:method name="get*" read-only="true"/>
      <!-- other methods use the default transaction settings (see below) -->
      <tx:method name="*"/>
    </tx:attributes>
  </tx:advice>

  <!-- ensure that the above transactional advice runs for any execution
      of an operation defined by the FooService interface -->
  <aop:config>
    <aop:pointcut id="fooServiceOperation" expression="execution(* x.y.service.FooService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceOperation"/>
  </aop:config>

  <!-- don't forget the DataSource -->
  <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
    <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
    <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
    <property name="username" value="scott"/>
    <property name="password" value="tiger"/>
  </bean>

  <!-- similarly, don't forget the PlatformTransactionManager -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

Let's pick apart the above configuration. We have a service object (the `'fooService'` bean) that we want to make transactional. The transaction semantics that we want to apply are encapsulated in the `<tx:advice/>` definition. The `<tx:advice/>` definition reads as "*... all methods on starting with `'get'` are to execute in the context of a read-only transaction, and all other methods are to execute with the default transaction semantics*". The `'transaction-manager'` attribute of the `<tx:advice/>` tag is set to the name of the `PlatformTransactionManager` bean that is going to actually *drive* the transactions (in this case the `'txManager'` bean).

Tip

You can actually omit the `'transaction-manager'` attribute in the transactional advice (`<tx:advice/>`) if the bean name of the `PlatformTransactionManager` that you want to wire in has the name `'transactionManager'`. If the `PlatformTransactionManager` bean that you want to wire in has any other name, then you have to be explicit and use the `'transaction-manager'` attribute as in the example above.

The `<aop:config/>` definition ensures that the transactional advice defined by the `'txAdvice'` bean actually executes at the appropriate points in the program. First we define a pointcut that matches the execution of any operation defined in the `FooService` interface (`'fooServiceOperation'`). Then we associate the pointcut with the `'txAdvice'` using an advisor. The result indicates that at the execution of a `'fooServiceOperation'`, the advice defined by `'txAdvice'` will be run.

The expression defined within the `<aop:pointcut/>` element is an AspectJ pointcut expression; see the chapter entitled Chapter 6, *Aspect Oriented Programming with Spring* for more details on pointcut expressions in

Spring 2.0.

A common requirement is to make an entire service layer transactional. The best way to do this is simply to change the pointcut expression to match any operation in your service layer. For example:

```
<aop:config>
    <aop:pointcut id="fooServiceMethods" expression="execution(* x.y.service.*.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="fooServiceMethods"/>
  </aop:config>
```

(This example assumes that all your service interfaces are defined in the `'x.y.service'` package; see the chapter entitled Chapter 6, Aspect Oriented Programming with Spring for more details.)

Now that we've analyzed the configuration, you may be asking yourself, "*Okay... but what does all this configuration actually do?*".

The above configuration is going to effect the creation of a transactional proxy around the object that is created from the `'fooService'` bean definition. The proxy will be configured with the transactional advice, so that when an appropriate method is invoked *on the proxy*, a transaction *may* be started, suspended, be marked as read-only, etc., depending on the transaction configuration associated with that method. Consider the following program that test drives the above configuration.

```
public final class Boot {

    public static void main(final String[] args) throws Exception {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("context.xml", Boot.class);
        FooService fooService = (FooService) ctx.getBean("fooService");
        fooService.insertFoo (new Foo());
    }
}
```

The output from running the above program will look something like this. *(Please note that the Log4J output and the stacktrace from the `UnsupportedOperationException` thrown by the `insertFoo(..)` method of the `DefaultFooService` class have been truncated in the interest of clarity.)*

```
    <!-- the Spring container is starting up... -->
[AspectJInvocationContextExposingAdvisorAutoProxyCreator] - Creating implicit proxy
        for bean 'fooService' with 0 common interceptors and 1 specific interceptors
    <!-- the DefaultFooService is actually proxied -->
[JdkDynamicAopProxy] - Creating JDK dynamic proxy for [x.y.service.DefaultFooService]

    <!-- ... the insertFoo(..) method is now being invoked on the proxy -->

[TransactionInterceptor] - Getting transaction for x.y.service.FooService.insertFoo
    <!-- the transactional advice kicks in here... -->
[DataSourceTransactionManager] - Creating new transaction with name [x.y.service.FooService.insertFoo]
[DataSourceTransactionManager] - Acquired Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4] for JDBC transaction

    <!-- the insertFoo(..) method from DefaultFooService throws an exception... -->
[RuleBasedTransactionAttribute] - Applying rules to determine whether transaction should
        rollback on java.lang.UnsupportedOperationException
[TransactionInterceptor] - Invoking rollback for transaction on x.y.service.FooService.insertFoo
        due to throwable [java.lang.UnsupportedOperationException]

   <!-- and the transaction is rolled back (by default, RuntimeException instances cause rollback) -->
[DataSourceTransactionManager] - Rolling back JDBC transaction on Connection
        [org.apache.commons.dbcp.PoolableConnection@a53de4]
[DataSourceTransactionManager] - Releasing JDBC Connection after transaction
[DataSourceUtils] - Returning JDBC Connection to DataSource

Exception in thread "main" java.lang.UnsupportedOperationException
        at x.y.service.DefaultFooService.insertFoo(DefaultFooService.java:14)
   <!-- AOP infrastructure stack trace elements removed for clarity -->
        at $Proxy0.insertFoo(Unknown Source)
        at Boot.main(Boot.java:11)
```

9.5.3. Rolling back

The previous section outlined the basics of how to specify the transactional settings for the classes, typically service layer classes, in your application in a declarative fashion. This section describes how you can control the rollback of transactions in a simple declarative fashion.

The recommended way to indicate to the Spring Framework's transaction infrastructure that a transaction's work is to be rolled back is to throw an `Exception` from code that is currently executing in the context of a transaction. The Spring Framework's transaction infrastructure code will catch any unhandled `Exception` as it bubbles up the call stack, and will mark the transaction for rollback.

However, please note that the Spring Framework's transaction infrastructure code will, by default, *only* mark a transaction for rollback in the case of runtime, unchecked exceptions; that is, when the thrown exception is an instance or subclass of `RuntimeException`. (`Errors` will also - by default - result in a rollback.) Checked exceptions that are thrown from a transactional method will *not* result in the transaction being rolled back.

Exactly which `Exception` types mark a transaction for rollback can be configured. Find below a snippet of XML configuration that demonstrates how one would configure rollback for a checked, application-specific `Exception` type.

```
<tx:advice id="txAdvice" transaction-manager="txManager">
  <tx:attributes>
        <tx:method name="get*" read-only="false" rollback-for="NoProductInStockException"/>
        <tx:method name="*"/>
  </tx:attributes>
</tx:advice>
```

The second way to indicate to the transaction infrastructure that a rollback is required is to do so *programmatically*. Although very simple, this way is quite invasive, and tightly couples your code to the Spring Framework's transaction infrastructure. Find below a snippet of code that does programmatic rollback of a Spring Framework-managed transaction:

```
public void resolvePosition() {
    try {
        // some business logic...
    } catch (NoProductInStockException ex) {
        // trigger rollback programmatically
        TransactionAspectSupport.currentTransactionStatus().setRollbackOnly();
    }
}
```

You are strongly encouraged to use the declarative approach to rollback if at all possible. Programmatic rollback is available should you need it, but its usage flies in the face of achieving a clean POJO-based application model.

9.5.4. Configuring different transactional semantics for different beans

Consider the scenario where you have a number of service layer objects, and you want to apply *totally different* transactional configuration to each of them. This is achieved by defining distinct `<aop:advisor/>` elements with differing `'pointcut'` and `'advice-ref'` attribute values.

Let's assume that all of your service layer classes are defined in a root `'x.y.service'` package. To make all beans that are instances of classes defined in that package (or in subpackages) and that have names ending in `'Service'` have the default transactional configuration, you would write the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
```

```
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <aop:config>

        <aop:pointcut id="serviceOperation"
                    expression="execution(* x.y.service..*Service.*(..))"/>

        <aop:advisor pointcut-ref="serviceOperation" advice-ref="txAdvice"/>

    </aop:config>

    <!-- these two beans will be transactional... -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>
    <bean id="barService" class="x.y.service.extras.SimpleBarService"/>

    <!-- ... and these two beans won't -->
    <bean id="anotherService" class="org.xyz.SomeService"/> <!-- (not in the right package) -->
    <bean id="barManager" class="x.y.service.SimpleBarManager"/> <!-- (doesn't end in 'Service') -->

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

Find below an example of configuring two distinct beans with totally different transactional settings.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:aop="http://www.springframework.org/schema/aop"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="
    http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
    http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
    http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <aop:config>

        <aop:pointcut id="defaultServiceOperation"
                    expression="execution(* x.y.service.*Service.*(..))"/>

        <aop:pointcut id="noTxServiceOperation"
                    expression="execution(* x.y.service.ddl.DefaultDdlManager.*(..))"/>

        <aop:advisor pointcut-ref="defaultServiceOperation" advice-ref="defaultTxAdvice"/>

        <aop:advisor pointcut-ref="noTxServiceOperation" advice-ref="noTxAdvice"/>

    </aop:config>

    <!-- this bean will be transactional (see the 'defaultServiceOperation' pointcut) -->
    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this bean will also be transactional, but with totally different transactional settings -->
    <bean id="anotherFooService" class="x.y.service.ddl.DefaultDdlManager"/>

    <tx:advice id="defaultTxAdvice">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <tx:advice id="noTxAdvice">
```

```
        <tx:attributes>
            <tx:method name="*" propagation="NEVER"/>
        </tx:attributes>
    </tx:advice>

    <!-- other transaction infrastructure beans such as a PlatformTransactionManager omitted... -->

</beans>
```

9.5.5. `<tx:advice/>` settings

This section summarises the various transactional settings that can be specified using the `<tx:advice/>` tag. The default `<tx:advice/>` settings are:

- The propagation setting is `REQUIRED`
- The isolation level is `DEFAULT`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any `RuntimeException` will trigger rollback, and any checked `Exception` will not

These default settings can, of course, be changed; the various attributes of the `<tx:method/>` tags that are nested within `<tx:advice/>` and `<tx:attributes/>` tags are summarized below:

Table 9.1. `<tx:method/>` settings

Attribute	Required?	Default	Description
`name`	Yes		The method name(s) with which the transaction attributes are to be associated. The wildcard (*) character can be used to associate the same transaction attribute settings with a number of methods; for example, `'get*'`, `'handle*'`, `'on*Event'`, etc.
`propagation`	No	REQUIRED	The transaction propagation behavior
`isolation`	No	DEFAULT	The transaction isolation level
`timeout`	No	-1	The transaction timeout value (in seconds)
`read-only`	No	false	Is this transaction

Attribute	Required?	Default	Description
			read-only?
`rollback-for`	No		The `Exception(s)` that will trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,Ser`
`no-rollback-for`	No		The `Exception(s)` that will *not* trigger rollback; comma-delimited. For example, `'com.foo.MyBusinessException,Ser`

At the time of writing it is not possible to have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method name of the transactionally-advised class. For example `'com.foo.BusinessService.handlePayment'`.

9.5.6. Using `@Transactional`

Note

The functionality offered by the `@Transactional` annotation and the support classes is only available to you if you are using Java 5+.

In addition to the XML-based declarative approach to transaction configuration, you can also use an annotation-based approach to transaction configuration. Declaring transaction semantics directly in the Java source code puts the declarations much closer to the affected code, and there is generally not much danger of undue coupling, since code that is meant to be used transactionally is almost always deployed that way anyway.

The ease-of-use afforded by the use of the `@Transactional` annotation is best illustrated with an example, after which all of the details will be explained. Consider the following class definition:

```
// the service class that we want to make transactional
@Transactional
public class DefaultFooService implements FooService {

    Foo getFoo(String fooName);

    Foo getFoo(String fooName, String barName);

    void insertFoo(Foo foo);

    void updateFoo(Foo foo);
}
```

When the above POJO is defined as a bean in a Spring IoC container, the bean instance can be made transactional by adding merely *one* line of XML configuration, like so:

```
<!-- from the file 'context.xml' -->
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- this is the service object that we want to make transactional -->
  <bean id="fooService" class="x.y.service.DefaultFooService"/>

  <!-- enable the configuration of transactional behavior based on annotations -->
  <tx:annotation-driven transaction-manager="txManager"/>

  <!-- a PlatformTransactionManager is still required -->
  <bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <!-- (this dependency is defined somewhere else) -->
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <!-- other <bean/> definitions here -->

</beans>
```

Tip

You can actually omit the `'transaction-manager'` attribute in the `<tx:annotation-driven/>` tag if the bean name of the `PlatformTransactionManager` that you want to wire in has the name `'transactionManager'`. If the `PlatformTransactionManager` bean that you want to dependency inject has any other name, then you have to be explicit and use the `'transaction-manager'` attribute as in the example above.

Method visibility and `@Transactional`

When using proxies, the `@Transactional` annotation should only be applied to methods with *public* visibility. If you do annotate protected, private or package-visible methods with the `@Transactional` annotation, no error will be raised, but the annotated method will not exhibit the configured transactional settings. Consider the use of AspectJ (see below) if you need to annotate non-public methods.

The `@Transactional` annotation may be placed before an interface definition, a method on an interface, a class definition, or a *public* method on a class. However, please note that the mere presence of the `@Transactional` annotation is not enough to actually turn on the transactional behavior - the `@Transactional` annotation *is simply metadata* that can be consumed by something that is `@Transactional`-aware and that can use the metadata to configure the appropriate beans with transactional behavior. In the case of the above example, it is the presence of the `<tx:annotation-driven/>` element that *switches on* the transactional behavior.

The Spring team's recommendation is that you only annotate concrete classes with the `@Transactional` annotation, as opposed to annotating interfaces. You certainly can place the `@Transactional` annotation on an interface (or an interface method), but this will only work as you would expect it to if you are using interface-based proxies. The fact that annotations are *not inherited* means that if you are using class-based proxies then the transaction settings will not be recognised by the class-based proxying infrastructure and the object will not be wrapped in a transactional proxy (which would be decidedly *bad*). So please do take the Spring team's advice and only annotate concrete classes (and the methods of concrete classes) with the `@Transactional` annotation.

Note: Since this mechanism is based on proxies, only 'external' method calls coming in through the proxy will be intercepted. This means that 'self-invocation', i.e. a method within the target object calling some other

method of the target object, won't lead to an actual transaction at runtime even if the invoked method is marked with `@Transactional`!

Table 9.2. `<tx:annotation-driven/>` settings

Attribute	Required?	Default	Description
`transaction-manager`	No	transactionManager	The name of transaction manager to use. Only required if the name of the transaction manager is not `transactionManager`, as in the example above.
`proxy-target-class`	No		Controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If "`proxy-target-class`" attribute is set to "`true`", then class-based proxies will be created. If "`proxy-target-class`" is "`false`" or if the attribute is omitted, then standard JDK interface-based proxies will be created. (See the section entitled Section 6.6, “Proxying mechanisms” for a detailed examination of the different proxy types.)
`order`	No		Defines the order of the transaction advice that will be applied to beans annotated with `@Transactional`. More on the rules related to ordering of AOP advice can be found in the AOP chapter (see section Section 6.2.4.7, “Advice ordering”). Note that not specifying any ordering will leave the decision as to what order advice is run in to the AOP subsystem.

Note

The "`proxy-target-class`" attribute on the `<tx:annotation-driven/>` element controls what type of transactional proxies are created for classes annotated with the `@Transactional` annotation. If "`proxy-target-class`" attribute is set to "`true`", then class-based proxies will be created. If "`proxy-target-class`" is "`false`" or if the attribute is omitted, then standard JDK interface-based proxies will be created. (See the section entitled Section 6.6, “Proxying mechanisms” for a detailed examination of the different proxy types.)

The most derived location takes precedence when evaluating the transactional settings for a method. In the case of the following example, the `DefaultFooService` class is annotated at the class level with the settings for a read-only transaction, but the `@Transactional` annotation on the `updateFoo(Foo)` method in the same class takes precedence over the transactional settings defined at the class level.

```
@Transactional(readOnly = true)
public class DefaultFooService implements FooService {

    public Foo getFoo(String fooName) {
        // do something
    }

    // these settings have precedence for this method
    @Transactional(readOnly = false, propagation = Propagation.REQUIRES_NEW)
    public void updateFoo(Foo foo) {
        // do something
    }
}
```

9.5.6.1. `@Transactional` settings

The `@Transactional` annotation is metadata that specifies that an interface, class, or method must have transactional semantics; for example, “*start a brand new read-only transaction when this method is invoked, suspending any existing transaction*”. The default `@Transactional` settings are:

- The propagation setting is `PROPAGATION_REQUIRED`
- The isolation level is `ISOLATION_DEFAULT`
- The transaction is read/write
- The transaction timeout defaults to the default timeout of the underlying transaction system, or or none if timeouts are not supported
- Any `RuntimeException` will trigger rollback, and any checked `Exception` will not

These default settings can, of course, be changed; the various properties of the `@Transactional` annotation are summarized in the following table:

Table 9.3. `@Transactional` properties

Property	Type	Description
`propagation`	enum: `Propagation`	optional propagation setting
`isolation`	enum: `Isolation`	optional isolation level

Property	Type	Description
`readOnly`	boolean	read/write vs. read-only transaction
`timeout`	int (in seconds granularity)	the transaction timeout
`rollbackFor`	an array of `Class` objects, which must be derived from `Throwable`	an optional array of exception classes which **must** cause rollback
`rollbackForClassname`	an array of class names. Classes must be derived from `Throwable`	an optional array of names of exception classes that **must** cause rollback
`noRollbackFor`	an array of `Class` objects, which must be derived from `Throwable`	an optional array of exception classes that **must not** cause rollback.
`noRollbackForClassname`	an array of `String` class names, which must be derived from `Throwable`	an optional array of names of exception classes that **must not** cause rollback

At the time of writing it is not possible to have explicit control over the name of a transaction, where 'name' means the transaction name that will be shown in a transaction monitor, if applicable (for example, WebLogic's transaction monitor), and in logging output. For declarative transactions, the transaction name is always the fully-qualified class name + "." + method name of the transactionally-advised class. For example `'com.foo.BusinessService.handlePayment'`.

9.5.7. Advising transactional operations

Consider the situation where you would like to execute *both* transactional *and* (to keep things simple) some basic profiling advice. Hhow do you effect this in the context of using `<tx:annotation-driven/>`?

What we want to see when we invoke the `updateFoo(Foo)` method is:

- the configured profiling aspect starting up,
- then the transactional advice executing,
- then the method on the advised object executing
- then the transaction committing (we'll assume a sunny day scenario here),
- and then finally the profiling aspect reporting (somehow) exactly how long the whole transactional method invocation took

Note

This chapter is not concerned with explaining AOP in any great detail (except as it applies to transactions). Please see the chapter entitled Chapter 6, *Aspect Oriented Programming with Spring* for detailed coverage of the various bits and pieces of the following AOP configuration (and AOP in general).

Here is the code for a simple profiling aspect. The ordering of advice is controlled via the `Ordered` interface. For full details on advice ordering, see Section 6.2.4.7, “Advice ordering”.

```
package x.y;

import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.util.StopWatch;
import org.springframework.core.Ordered;

public class SimpleProfiler implements Ordered {

    private int order;

    // allows us to control the ordering of advice
    public int getOrder() {
        return this.order;
    }

    public void setOrder(int order) {
        this.order = order;
    }

    // this method is the around advice
    public Object profile(ProceedingJoinPoint call) throws Throwable {
        Object returnValue;
        StopWatch clock = new StopWatch(getClass().getName());
        try {
            clock.start(call.toShortString());
            returnValue = call.proceed();
        } finally {
            clock.stop();
            System.out.println(clock.prettyPrint());
        }
        return returnValue;
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- this is the aspect -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <tx:annotation-driven transaction-manager="txManager" order="200"/>

    <aop:config>
        <!-- this advice will execute around the transactional advice -->
        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                    expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>
    </aop:config>

    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="oracle.jdbc.driver.OracleDriver"/>
        <property name="url" value="jdbc:oracle:thin:@rj-t42:1521:elvis"/>
        <property name="username" value="scott"/>
        <property name="password" value="tiger"/>
    </bean>

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

</beans>
```

The result of the above configuration will be a `'fooService'` bean that has profiling and transactional aspects applied to it *in that order*. The configuration of any number of additional aspects is effected in a similar fashion.

Finally, find below some example configuration for effecting the same setup as above, but using the purely XML declarative approach.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

    <bean id="fooService" class="x.y.service.DefaultFooService"/>

    <!-- the profiling advice -->
    <bean id="profiler" class="x.y.SimpleProfiler">
        <!-- execute before the transactional advice (hence the lower order number) -->
        <property name="order" value="1"/>
    </bean>

    <aop:config>

        <aop:pointcut id="entryPointMethod" expression="execution(* x.y..*Service.*(..))"/>

        <!-- will execute after the profiling advice (c.f. the order attribute) -->
        <aop:advisor
                advice-ref="txAdvice"
                pointcut-ref="entryPointMethod"
                order="2"/> <!-- order value is higher than the profiling aspect -->

        <aop:aspect id="profilingAspect" ref="profiler">
            <aop:pointcut id="serviceMethodWithReturnValue"
                          expression="execution(!void x.y..*Service.*(..))"/>
            <aop:around method="profile" pointcut-ref="serviceMethodWithReturnValue"/>
        </aop:aspect>

    </aop:config>

    <tx:advice id="txAdvice" transaction-manager="txManager">
        <tx:attributes>
            <tx:method name="get*" read-only="true"/>
            <tx:method name="*"/>
        </tx:attributes>
    </tx:advice>

    <!-- other <bean/> definitions such as a DataSource and a PlatformTransactionManager here -->

</beans>
```

The result of the above configuration will be a `'fooService'` bean that has profiling and transactional aspects applied to it *in that order*. If we wanted the profiling advice to execute *after* the transactional advice on the way in, and *before* the transactional advice on the way out, then we would simply swap the value of the profiling aspect bean's `'order'` property such that it was higher than the transactional advice's order value.

The configuration of any number of additional aspects is achieved in a similar fashion.

9.5.8. Using `@Transactional` with AspectJ

It is also possible to use the Spring Framework's `@Transactional` support outside of a Spring container by means of an AspectJ aspect. To use this support you must first annotate your classes (and optionally your classes' methods with the `@Transactional` annotation, and then you must link (weave) your application with the `org.springframework.transaction.aspectj.AnnotationTransactionAspect` defined in the

`spring-aspects.jar` file. The aspect must also be configured with a transaction manager. You could of course use the Spring Framework's IoC container to take care of dependency injecting the aspect, but since we're focusing here on applications running outside of a Spring container, we'll show you how to do it programmatically.

Note

Prior to continuing, you may well want to read the previous sections entitled Section 9.5.6, “Using `@Transactional`” and Chapter 6, *Aspect Oriented Programming with Spring* respectively.

```
// construct an appropriate transaction manager
DataSourceTransactionManager txManager = new DataSourceTransactionManager(getDataSource());

// configure the AnnotationTransactionAspect to use it; this must be done before executing any transactional methods
AnnotationTransactionAspect.aspectOf().setTransactionManager(txManager);
```

Note

When using this aspect, you must annotate the *implementation* class (and/or methods within that class), *not* the interface (if any) that the class implements. AspectJ follows Java's rule that annotations on interfaces are *not inherited*.

The `@Transactional` annotation on a class specifies the default transaction semantics for the execution of any method in the class.

The `@Transactional` annotation on a method within the class overrides the default transaction semantics given by the class annotation (if present). Any method may be annotated, regardless of visibility.

To weave your applications with the `AnnotationTransactionAspect` you must either build your application with AspectJ (see the AspectJ Development Guide) or use load-time weaving. See the section entitled Section 6.8.4, “Using AspectJ Load-time weaving (LTW) with Spring applications” for a discussion of load-time weaving with AspectJ.

9.6. Programmatic transaction management

The Spring Framework provides two means of programmatic transaction management:

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

If you are going to use programmatic transaction management, the Spring team generally recommend, namely that of using the `TransactionTemplate`). The second approach is similar to using the JTA `UserTransaction` API (although exception handling is less cumbersome).

9.6.1. Using the `TransactionTemplate`

The `TransactionTemplate` adopts the same approach as other Spring *templates* such as the `JdbcTemplate`. It uses a callback approach, to free application code from having to do the boilerplate acquisition and release of transactional resources, and results in code that is intention driven, in that the code that is written focuses solely on what the developer wants to do.

Note

As you will immediately see in the examples that follow, using the `TransactionTemplate` absolutely couples you to Spring's transaction infrastructure and APIs. Whether or not programmatic transaction management is suitable for your development needs is a decision that you will have to make yourself.

Application code that must execute in a transactional context, and that will use the `TransactionTemplate` explicitly, looks like this. You, as an application developer, will write a `TransactionCallback` implementation (typically expressed as an anonymous inner class) that will contain all of the code that you need to have execute in the context of a transaction. You will then pass an instance of your custom `TransactionCallback` to the `execute(..)` method exposed on the `TransactionTemplate`.

```
public class SimpleService implements Service {

    // single TransactionTemplate shared amongst all methods in this instance
    private final TransactionTemplate transactionTemplate;

    // use constructor-injection to supply the PlatformTransactionManager
    public SimpleService(PlatformTransactionManager transactionManager) {
        Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
        this.transactionTemplate = new TransactionTemplate(transactionManager);
    }

    public Object someServiceMethod() {
        return transactionTemplate.execute(new TransactionCallback() {

            // the code in this method executes in a transactional context
            public Object doInTransaction(TransactionStatus status) {
                updateOperation1();
                return resultOfUpdateOperation2();
            }
        });
    }
}
```

If there is no return value, use the convenient `TransactionCallbackWithoutResult` class via an anonymous class like so:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        updateOperation1();
        updateOperation2();
    }
});
```

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

```
transactionTemplate.execute(new TransactionCallbackWithoutResult() {

    protected void doInTransactionWithoutResult(TransactionStatus status) {
        try {
            updateOperation1();
            updateOperation2();
        } catch (SomeBusinessExeption ex) {
            status.setRollbackOnly();
        }
    }
});
```

9.6.1.1. Specifying transaction settings

Transaction settings such as the propagation mode, the isolation level, the timeout, and so forth can be set on the `TransactionTemplate` either programmatically or in configuration. `TransactionTemplate` instances by default have the default transactional settings. Find below an example of programmatically customizing the transactional settings for a specific `TransactionTemplate`.

```
public class SimpleService implements Service {

    private final TransactionTemplate transactionTemplate;

    public SimpleService(PlatformTransactionManager transactionManager) {
        Assert.notNull(transactionManager, "The 'transactionManager' argument must not be null.");
        this.transactionTemplate = new TransactionTemplate(transactionManager);

        // the transaction settings can be set here explicitly if so desired
        this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATION_READ_UNCOMMITTED);
        this.transactionTemplate.setTimeout(30); // 30 seconds
        // and so forth...
    }
}
```

Find below an example of defining a `TransactionTemplate` with some custom transactional settings, using Spring XML configuration. The '`sharedTransactionTemplate`' can then be injected into as many services as are required.

```
<bean id="sharedTransactionTemplate"
      class="org.springframework.transaction.support.TransactionTemplate">
    <property name="isolationLevelName" value="ISOLATION_READ_UNCOMMITTED"/>
    <property name="timeout" value="30"/>
</bean>"
```

Finally, instances of the `TransactionTemplate` class are threadsafe, in that instances do not maintain any conversational state. `TransactionTemplate` instances *do* however maintain configuration state, so while a number of classes may choose to share a single instance of a `TransactionTemplate`, if a class needed to use a `TransactionTemplate` with different settings (for example, a different isolation level), then two distinct `TransactionTemplate` instances would need to be created and used.

9.6.2. Using the `PlatformTransactionManager`

You can also use the `org.springframework.transaction.PlatformTransactionManager` directly to manage your transaction. Simply pass the implementation of the `PlatformTransactionManager` you're using to your bean via a bean reference. Then, using the `TransactionDefinition` and `TransactionStatus` objects you can initiate transactions, rollback and commit.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
// explicitly setting the transaction name is something that can only be done programmatically
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);
try {
    // execute your business logic here
}
catch (MyException ex) {
    txManager.rollback(status);
    throw ex;
}
txManager.commit(status);
```

9.7. 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. In this case, using the `TransactionTemplate` *may* be a good approach. Being able to set the transaction name explicitly is also something that can only be done using the programmatic approach to transaction management.

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

9.8. Application server-specific integration

Spring's transaction abstraction is generally application server agnostic. Additionally, Spring's `JtaTransactionManager` class, which can optionally perform a JNDI lookup for the JTA `UserTransaction` and `TransactionManager` objects, can be set to autodetect the location for the latter object, which varies by application server. Having access to the `TransactionManager` instance does allow enhanced transaction semantics. Please see the `JtaTransactionManager` Javadocs for more details.

9.8.1. BEA WebLogic

In a WebLogic 7.0, 8.1 or higher environment, you will generally prefer to use `WebLogicJtaTransactionManager` instead of the stock `JtaTransactionManager` class. This special WebLogic-specific subclass of the normal `JtaTransactionManager`. It supports the full power of Spring's transaction definitions in a WebLogic managed transaction environment, beyond standard JTA semantics: features include transaction names, per-transaction isolation levels, and proper resuming of transactions in all cases.

9.8.2. IBM WebSphere

In a WebSphere 5.1, 5.0 and 4 environment, you may wish to use Spring's `WebSphereTransactionManagerFactoryBean` class. This is a factory bean which retrieves the JTA `TransactionManager` in a WebSphere environment, which is done via WebSphere's `static` access methods. (These methods are different for each version of WebSphere.) Once the JTA `TransactionManager` instance has been obtained via this factory bean, Spring's `JtaTransactionManager` may be configured with a reference to it, for enhanced transaction semantics over the use of only the JTA `UserTransaction` object. Please see the Javadocs for full details.

9.9. Solutions to common problems

9.9.1. Use of the wrong transaction manager for a specific `DataSource`

You should take care to use the correct *`PlatformTransactionManager`* implementation for their requirements. It is important to understand how the the Spring Framework's transaction abstraction works with JTA global transactions. Used properly, there is no conflict here: the Spring Framework merely provides a straightforward

and portable abstraction. If you are using global transactions, you *must* use the `org.springframework.transaction.jta.JtaTransactionManager` class (or an application server-specific subclass of it) for all your transactional operations. Otherwise the transaction infrastructure will attempt to perform local transactions on resources such as container `DataSource` instances. Such local transactions don't make sense, and a good application server will treat them as errors.

9.10. Further Resources

Find below links to further resources about the Spring Framework's transaction support.

- This book, Java Transaction Design Strategies from InfoQ is a well-paced introduction to transactions in the Java space. It also includes side-by-side examples of how to configure and use transactions in both the Spring Framework and EJB3.

Chapter 10. DAO support

10.1. Introduction

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

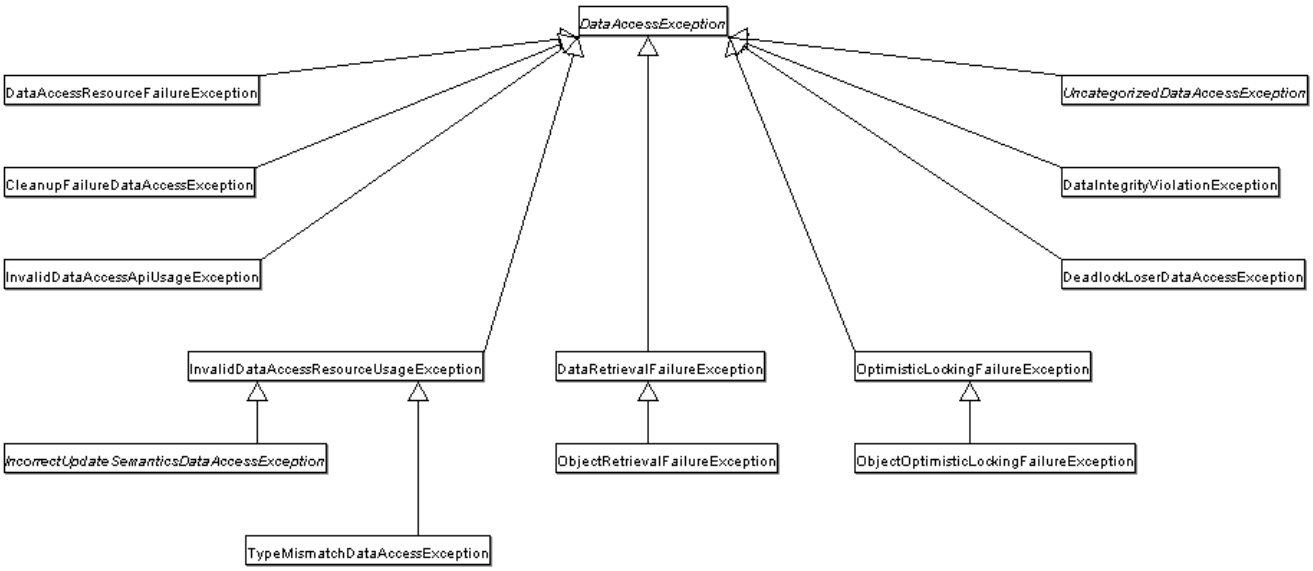
10.2. Consistent exception hierarchy

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

In addition to JDBC exceptions, Spring can also wrap Hibernate-specific exceptions, converting them from proprietary, checked exceptions (in the case of versions of Hibernate prior to Hibernate 3.0), to a set of focused runtime exceptions (the same is true for JDO and JPA exceptions). This allows one to handle most persistence exceptions, which are non-recoverable, only in the appropriate layers, without having annoying boilerplate catch-and-throw blocks and exception declarations in one's DAOs. (One can still trap and handle exceptions anywhere one needs to though.) As mentioned above, JDBC exceptions (including database-specific dialects) are also converted to the same hierarchy, meaning that one can perform some operations with JDBC within a consistent programming model.

The above holds true for the various template classes in Springs support for various ORM frameworks. If one uses 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 convert the exceptions to ones that are compatible with the exceptions in 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 provides can be seen below. (Please note that the class hierarchy detailed in the image shows only a subset of the entire `DataAccessException` hierarchy.)



10.3. Consistent abstract classes for DAO support

To make it easier to work with a variety of data access technologies such as JDBC, JDO and Hibernate in a consistent way, Spring provides a set of `abstract` DAO classes that one can extend. These abstract classes have methods for providing the data source and any other configuration settings that are specific to the relevant data-access technology.

- `JdbcDaoSupport` - superclass for JDBC data access objects. Requires a `DataSource` to be provided; in turn, this class provides a `JdbcTemplate` instance initialized from the supplied `DataSource` to subclasses.
- `HibernateDaoSupport` - superclass for Hibernate data access objects. Requires a `SessionFactory` to be provided; in turn, this class provides a `HibernateTemplate` instance initialized from the supplied `SessionFactory` to subclasses. Can alternatively be initialized directly via a `HibernateTemplate`, to reuse the latters settings like `SessionFactory`, flush mode, exception translator, and so forth.
- `JdoDaoSupport` - super class for JDO data access objects. Requires a `PersistenceManagerFactory` to be provided; in turn, this class provides a `JdoTemplate` instance initialized from the supplied `PersistenceManagerFactory` to subclasses.
- `JpaDaoSupport` - super class for JPA data access objects. Requires a `EntityManagerFactory` to be provided; in turn, this class provides a `JpaTemplate` instance initialized from the supplied `EntityManagerFactory` to subclasses.

Chapter 11. Data access using JDBC

11.1. Introduction

The value-add provided by the Spring Framework's JDBC abstraction framework is perhaps best shown by the following list (note that only the italicized lines need to be coded by an application developer):

1. Define connection parameters
2. Open the connection
3. *Specify the statement*
4. Prepare and execute the statement
5. Set up the loop to iterate through the results (if any)
6. *Do the work for each iteration*
7. Process any exception
8. Handle transactions
9. Close the connection

The Spring Framework takes care of all the grungy, low-level details that can make JDBC such a tedious API to develop with.

11.1.1. The package hierarchy

The Spring Framework's JDBC abstraction framework consists of four different packages, namely `core`, `dataSource`, `object`, and `support`.

The `org.springframework.jdbc.core` package contains the `JdbcTemplate` class and its various callback interfaces, plus a variety of related classes.

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 thread safe, reusable objects. This approach is modeled 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.

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

11.2.1. `JdbcTemplate`

The `JdbcTemplate` class 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 such as 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 the `JdbcTemplate` 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 `CallableStatementCreator` interface which creates callable statement. The `RowCallbackHandler` interface extracts values from each row of a `ResultSet`.

The `JdbcTemplate` can be used within a DAO implementation via direct instantiation with a `DataSource` reference, or be configured in a Spring IOC container and given to DAOs as a bean reference. Note: the `DataSource` should always be configured as a bean in the Spring IoC container, in the first case given to the service directly, in the second case to the prepared template.

Finally, all of the SQL issued by this class is logged at the `'DEBUG'` level under the category corresponding to the fully qualified class name of the template instance (typically `JdbcTemplate`, but it may be different if a custom subclass of the `JdbcTemplate` class is being used).

11.2.1.1. Examples

Find below some examples of using the `JdbcTemplate` class. (These examples are not an exhaustive list of all of the functionality exposed by the `JdbcTemplate`; see the attendant Javadocs for that).

11.2.1.1.1. Querying (SELECT)

A simple query for getting the number of rows in a relation.

```
int rowCount = this.jdbcTemplate.queryForInt("select count(0) from t_accrual");
```

A simple query using a bind variable.

```
int countOfActorsNamedJoe
    = this.jdbcTemplate.queryForInt("select count(0) from t_actors where first_name = ?", new Object[]{"Joe"});
```

Querying for a `String`.

```
String surname = (String) this.jdbcTemplate
    .queryForObject("select surname from t_actor where id = ?", new Object[]{new Long(1212)}, String.class);
```

Querying and populating a *single* domain object.

```
Actor actor = (Actor) this.jdbcTemplate.queryForObject(
    "select first_name, surname from t_actor where id = ?",
    new Object[]{new Long(1212)},
    new RowMapper() {

        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setFirstName(rs.getString("first_name"));
            actor.setSurname(rs.getString("surname"));
            return actor;
        }
    });
```

Querying and populating a number of domain objects.

```
Collection actors = this.jdbcTemplate.query(
    "select first_name, surname from t_actor",
    new RowMapper() {

        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setFirstName(rs.getString("first_name"));
            actor.setSurname(rs.getString("surname"));
            return actor;
        }
    });
```

If the last two snippets of code actually existed in the same application, it would make sense to remove the duplication present in the two `RowMapper` anonymous inner classes, and extract them out into a single class (typically a `static` inner class) that can then be referenced by DAO methods as needed. For example, the last code snippet might be better off written like so:

```
public Collection findAllActors() {
    return this.jdbcTemplate.query( "select first_name, surname from t_actor", new ActorMapper());
}

private static final class ActorMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Actor actor = new Actor();
        actor.setFirstName(rs.getString("first_name"));
        actor.setSurname(rs.getString("surname"));
        return actor;
    }
}
```

11.2.1.1.2. Updating (INSERT/UPDATE/DELETE)

```
this.jdbcTemplate.update("insert into t_actor (first_name, surname) values (?, ?)", new Object[] {"Leonor", "Wat
```

```
this.jdbcTemplate.update("update t_actor set weapon = ? where id = ?", new Object[] {"Banjo", new Long(5276)});
```

```
this.jdbcTemplate.update("delete from orders"); // :)
```

11.2.1.1.3. Other operations

The `execute(..)` method can be used to execute any arbitrary SQL, and as such is often used for DDL statements. It is heavily overloaded with variants taking callback interfaces, bind variable arrays, and suchlike.

```
this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
```

Invoking a simple stored procedure (more sophisticated stored procedure support is covered later).

```
this.jdbcTemplate.update("call SUPPORT.REFRESH_ACTORS_SUMMARY(?)", new Object[]{new Long(unionId)});
```

11.2.1.2. `JdbcTemplate` idioms (best practices)

Instances of the `JdbcTemplate` class are *threadsafe once configured*. This is important because it means that you can configure a single instance of a `JdbcTemplate` and then safely inject this *shared* reference into multiple DAOs (or repositories). To be clear, the `JdbcTemplate` is stateful, in that it maintains a reference to a `DataSource`, but this state is *not* conversational state.

A common idiom when using the `JdbcTemplate` class (and the associated `SimpleJdbcTemplate` and `NamedParameterJdbcTemplate` classes) is to configure a `DataSource` in your Spring configuration file, and then dependency inject that shared `DataSource` bean into your DAO classes; the `JdbcTemplate` is created in the setter for the `DataSource`. This leads to DAOs that look in part like this:

```
public class JdbcCorporateEventDao implements CorporateEventDao {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // JDBC-backed implementations of the methods on the CorporateEventDao follow...
}
```

The attendant configuration might look like this.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/bea

    <bean id="corporateEventDao" class="com.example.JdbcCorporateEventDao">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!-- the DataSource (parameterized for configuration via a PropertyPlaceHolderConfigurer) -->
    <bean id="dataSource" destroy-method="close" class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="${jdbc.driverClassName}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

</beans>
```

If you are using Spring's `JdbcDaoSupport` class, and your various JDBC-backed DAO classes extend from it, then you inherit a `setDataSource(..)` method for free from said superclass. It is totally up to you as to whether or not you inherit from said class, you certainly are not forced to. If you look at the source for the `JdbcDaoSupport` class you will see that there is not a whole lot to it... it is provided as a convenience only.

Regardless of which of the above template initialization styles you choose to use (or not), there is (almost) certainly no need to create a brand new instance of a `JdbcTemplate` class each and every time you wish to execute some SQL... remember, once configured, a `JdbcTemplate` instance is threadsafe. A reason for wanting multiple `JdbcTemplate` instances would be when you have an application that accesses multiple databases, which requires multiple `DataSources`, and subsequently multiple differently configured `JdbcTemplates`.

11.2.2. `NamedParameterJdbcTemplate`

The `NamedParameterJdbcTemplate` class adds support for programming JDBC statements using named parameters (as opposed to programming JDBC statements using only classic placeholder (`'?'`) arguments. The `NamedParameterJdbcTemplate` class wraps a `JdbcTemplate`, and delegates to the wrapped `JdbcTemplate` to do much of its work. This section will describe only those areas of the `NamedParameterJdbcTemplate` class that differ from the `JdbcTemplate` itself; namely, programming JDBC statements using named parameters.

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(0) from T_ACTOR where first_name = :first_name";

    SqlParameterSource namedParameters = new MapSqlParameterSource("first_name", firstName);

    return namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```

Notice the use of the named parameter notation in the value assigned to the `'sql'` variable, and the corresponding value that is plugged into the `'namedParameters'` variable (of type `MapSqlParameterSource`).

If you like, you can also pass along named parameters (and their corresponding values) to a `NamedParameterJdbcTemplate` instance using the (perhaps more familiar) `Map`-based style. (The rest of the methods exposed by the `NamedParameterJdbcOperations` - and implemented by the `NamedParameterJdbcTemplate` class) follow a similar pattern and will not be covered here.)

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActorsByFirstName(String firstName) {

    String sql = "select count(0) from T_ACTOR where first_name = :first_name";

    Map namedParameters = Collections.singletonMap("first_name", firstName);

    return this.namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```

Another nice feature related to the `NamedParameterJdbcTemplate` (and existing in the same Java package) is the `SqlParameterSource` interface. You have already seen an example of an implementation of this interface in one of the preceding code snippets (the `MapSqlParameterSource` class). The entire point of the `SqlParameterSource` is to serve as a source of named parameter values to a `NamedParameterJdbcTemplate`. The `MapSqlParameterSource` class is a very simple implementation, that is simply an adapter around a `java.util.Map`, where the keys are the paramter names and the values are the parameter values.

Another `SqlParameterSource` implementation is the `BeanPropertySqlParameterSource` class. This class wraps an arbitrary JavaBean (that is, an instance of a class that adheres to the JavaBean conventions), and uses the properties of the wrapped JavaBean as the source of named parameter values.

```
public class Actor {

    private Long id;
    private String firstName;
    private String lastName;

    public String getFirstName() {
```

```
        return this.firstName;
    }

    public String getLastName() {
        return this.lastName;
    }

    public Long getId() {
        return this.id;
    }

    // setters omitted...

}
```

```
// some JDBC-backed DAO class...
private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.namedParameterJdbcTemplate = new NamedParameterJdbcTemplate(dataSource);
}

public int countOfActors(Actor exampleActor) {

    // notice how the named parameters match the properties of the above 'Actor' class
    String sql = "select count(0) from T_ACTOR where first_name = :firstName and last_name = :lastName";

    SqlParameterSource namedParameters = new BeanPropertySqlParameterSource(exampleActor);

    return this.namedParameterJdbcTemplate.queryForInt(sql, namedParameters);
}
```

Remember that the `NamedParameterJdbcTemplate` class *wraps* a classic `JdbcTemplate` template; if you need access to the wrapped `JdbcTemplate` instance (to access some of the functionality only present in the `JdbcTemplate` class), then you can use the `getJdbcOperations()` method to access the wrapped `JdbcTemplate` *via the `JdbcOperations` interface*.

See also the section entitled Section 11.2.1.2, “`JdbcTemplate` idioms (best practices)” for some advice on how to best use the `NamedParameterJdbcTemplate` class in the context of an application.

11.2.3. `SimpleJdbcTemplate`

Note

The functionality offered by the `SimpleJdbcTemplate` is only available to you if you are using Java 5.

The `SimpleJdbcTemplate` class is a wrapper around the classic `JdbcTemplate` that takes advantage of Java 5 language features such as varargs and autoboxing. The `SimpleJdbcTemplate` class is somewhat of a sop to the syntactic-sugar-like features of Java 5, but as anyone who has developed on Java 5 and then had to move back to developing on a previous version of the JDK will know, those syntactic-sugar-like features sure are nice.

The value-add of the `SimpleJdbcTemplate` class in the area of syntactic-sugar is best illustrated with a *'before and after'* example. The following code snippet shows first some data access code using the classic `JdbcTemplate`, followed immediately thereafter by a code snippet that does the same job, only this time using the `SimpleJdbcTemplate`.

```
// classic JdbcTemplate-style...
private JdbcTemplate jdbcTemplate;

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

```
}

public Actor findActor(long id) {
    String sql = "select id, first_name, last_name from T_ACTOR where id = ?";

    RowMapper mapper = new RowMapper() {

        public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong("id"));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    };

    // notice the cast, the wrapping up of the 'id' argument
    // in an array, and the boxing of the 'id' argument as a reference type
    return (Actor) jdbcTemplate.queryForObject(sql, mapper, new Object[] {Long.valueOf(id)});
}
```

Here is the same method, only this time using the `SimpleJdbcTemplate`; notice how much 'cleaner'the code is.

```
// SimpleJdbcTemplate-style...
private SimpleJdbcTemplate simpleJdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.simpleJdbcTemplate = new SimpleJdbcTemplate(dataSource);
}
public Actor findActor(long id) {
    String sql = "select id, first_name, last_name from T_ACTOR where id = ?";

    ParameterizedRowMapper<Actor> mapper = new ParameterizedRowMapper<Actor>() {

        // notice the return type with respect to Java 5 covariant return types
        public Actor mapRow(ResultSet rs, int rowNum) throws SQLException {
            Actor actor = new Actor();
            actor.setId(rs.getLong("id"));
            actor.setFirstName(rs.getString("first_name"));
            actor.setLastName(rs.getString("last_name"));
            return actor;
        }
    };

    return this.simpleJdbcTemplate.queryForObject(sql, mapper, id);
}
```

See also the section entitled Section 11.2.1.2, “`JdbcTemplate` idioms (best practices)” for some advice on how to best use the `SimpleJdbcTemplate` class in the context of an application.

Note

The `SimpleJdbcTemplate` class only offers a much smaller subset of the methods exposed on the `JdbcTemplate` class. If you need to use a method from the `JdbcTemplate` that is not defined on the `SimpleJdbcTemplate`, you can always access the underlying `JdbcTemplate` by calling the `getJdbcOperations()` method on the `SimpleJdbcTemplate`, which will then allow you to invoke the method that you want. The only downside is that the methods on the `JdbcOperations` interface are not generified, so you are back to casting and such again.

11.2.4. `DataSource`

In order to work with data from a database, one needs 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 not 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 fulfill 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("");
```

11.2.5. `SQLExceptionTranslator`

`SQLExceptionTranslator` is an interface to be implemented by classes that can translate between `SQLExceptions` and Spring's own 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 does not 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 MySQLErrorCodesTranslator 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`.

11.2.6. 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 jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void doExecute() {
        this.jdbcTemplate.execute("create table mytable (id integer, name varchar(100))");
    }
}
```

11.2.7. 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 jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
```

```
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public int getCount() {
        return this.jdbcTemplate.queryForInt("select count(*) from mytable");
    }

    public String getName() {
        return (String) this.jdbcTemplate.queryForObject("select name from mytable", String.class);
    }

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

In addition to the single results query methods there are several methods that return a List with an entry for each row that the query returned. The most generic method 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:

```
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.jdbcTemplate = new JdbcTemplate(dataSource);
}

public List getList() {
    return this.jdbcTemplate.queryForList("select * from mytable");
}
```

The list returned would look something like this:

```
[{name=Bob, id=1}, {name=Mary, id=2}]
```

11.2.8. Updating the database

There are also a number of update methods that you can use. Find below an example where a column is updated for a certain primary key. In this example an SQL statement is used that has placeholders for row parameters. Note that the parameter values are passed in as an array of objects (and thus primitives have to be wrapped in the primitive wrapper classes).

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate jdbcTemplate;

    public void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    public void setName(int id, String name) {
        this.jdbcTemplate.update("update mytable set name = ? where id = ?", new Object[] {name, new Integer(id)
    }
}
```

One of the `update` convenience methods provides support for acquiring the primary keys generated by the database (part of the JDBC 3.0 standard - see chapter 13.6 of the specification for details). The method takes a `PreparedStatementCreator` as its first argument, and this is the way the required insert statement is specified.

The other argument is a `KeyHolder`, which will contain the generated key on successful return from the update. There is not a standard single way to create an appropriate `PreparedStatement` (which explains why the method signature is the way it is). An example that works on Oracle and may work on other platforms is:

```
final String INSERT_SQL = "insert into my_test (name) values(?)";
final String name = "Rob";

KeyHolder keyHolder = new GeneratedKeyHolder();
jdbcTemplate.update(
    new PreparedStatementCreator() {
        public PreparedStatement createPreparedStatement(Connection connection) throws SQLException {
            PreparedStatement ps =
                connection.prepareStatement(INSERT_SQL, new String[] {"id"});
            ps.setString(1, name);
            return ps;
        }
    },
    keyHolder);

// keyHolder.getKey() now contains the generated key
```

11.3. Controlling database connections

11.3.1. `DataSourceUtils`

The `DataSourceUtils` class is a convenient and powerful helper class that provides `static` methods to obtain connections from JNDI and close connections if necessary. It has support for thread-bound connections, for example for use with `DataSourceTransactionManager`.

11.3.2. `SmartDataSource`

The `SmartDataSource` interface is to be implemented by classes that can provide a connection to a relational database. Extends the `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, in the cases where one knows that one wants to reuse a connection.

11.3.3. `AbstractDataSource`

This is an `abstract` base class for Spring's `DataSource` implementations, that takes care of the "uninteresting" glue. This is the class one would extend if one was writing one's own `DataSource` implementation.

11.3.4. `SingleConnectionDataSource`

The `SingleConnectionDataSource` class is an implementation of the `SmartDataSource` interface that wraps a *single* `Connection` that 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.

11.3.5. `DriverManagerDataSource`

The `DriverManagerDataSource` class is an implementation of the `SmartDataSource` interface that configures a plain old JDBC Driver via bean properties, and returns a new connection every time.

This is potentially useful for test or standalone environments outside of a J2EE container, either as a `DataSource` bean in a Spring IoC container, 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. However, using JavaBean style connection pools such as commons-dbcp is so easy, even in a test environment, that it is almost always preferable to use such a connection pool over `DriverManagerDataSource`.

11.3.6. `TransactionAwareDataSourceProxy`

`TransactionAwareDataSourceProxy` is a proxy for a target `DataSource`, which wraps that target `DataSource` to add awareness of Spring-managed transactions. In this respect it is similar to a transactional JNDI `DataSource` as provided by a J2EE server.

Note

It should almost never be necessary or desireable to use this class, except when existing code exists which must be called and passed a standard JDBC `DataSource` interface implementation. In this case, it's possible to still have this code be usable, but participating in Spring managed transactions. It is generally preferable to write your own new code using the higher level abstractions for resource management, such as `JdbcTemplate` or `DataSourceUtils`.

(See the `TransactionAwareDataSourceProxy` Javadocs for more details.)

11.3.7. `DataSourceTransactionManager`

The `DataSourceTransactionManager` class is a `PlatformTransactionManager` implementation for single JDBC datasources. It binds a JDBC connection from the specified data source to the currently executing 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 `SQLExceptions`. 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.

The `DataSourceTransactionManager` class 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!

11.4. Modeling JDBC operations as Java objects

The `org.springframework.jdbc.object` package contains classes that allow one to access the database in a

more object-oriented manner. By way of an example, one 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. One can also execute stored procedures and run update, delete and insert statements.

Note

There is a view borne from experience acquired in the field amongst some of the Spring developers that the various RDBMS operation classes described below (with the exception of the `StoredProcedure` class) can often be replaced with straight `JdbcTemplate` calls... often it is simpler to use and plain easier to read a DAO method that simply calls a method on a `JdbcTemplate` direct (as opposed to encapsulating a query as a full-blown class).

It must be stressed however that this is just a *view*... if you feel that you are getting measurable value from using the RDBMS operation classes, feel free to continue using these classes.

11.4.1. `SqlQuery`

`SqlQuery` is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the `newRowMapper(..)` method to provide a `RowMapper` instance that can create one object per row obtained from iterating over the `ResultSet` that is created during the execution of the query. The `SqlQuery` class is rarely used directly since the `MappingSqlQuery` subclass provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend `SqlQuery` are `MappingSqlQueryWithParameters` and `UpdatableSqlQuery`.

11.4.2. `MappingSqlQuery`

`MappingSqlQuery` is a reusable query in which concrete subclasses must implement the abstract `mapRow(..)` method to convert each row of the supplied `ResultSet` into an object. Find below a brief example of a custom query that maps the data from the customer relation to an instance of the `Customer` class.

```
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.

```
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.

11.4.3. `SqlUpdate`

The `SqlUpdate` class encapsulates an SQL update. Like a query, an update object is reusable, and like all `RdbmsOperation` classes, 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);
    }
}
```

11.4.4. `StoredProcedure`

The `StoredProcedure` class is a superclass for object abstractions of RDBMS stored procedures. This class is `abstract`, and its various `execute(..)` methods have `protected` access, preventing use other than through a subclass that offers tighter typing.

The inherited `sql` property will be 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 one has to create a class that extends `StoredProcedure`. There are no input parameters, but there is an output parameter that is declared as a date type 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 TestStoredProcedure {

    public static void main(String[] args)  {
        TestStoredProcedure t = new TestStoredProcedure();
        t.test();
        System.out.println("Done!");
    }

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

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

    private class MyStoredProcedure extends StoredProcedure {

        private 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() {
            // the 'sysdate' sproc has no input parameters, so an empty Map is supplied...
            return execute(new HashMap());
        }
    }

    private static void printMap(Map results) {
        for (Iterator it = results.entrySet().iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
    }
}
```

Find below an example of a `StoredProcedure` that has two output parameters (in this case Oracle cursors).

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAndGenresStoredProcedure extends StoredProcedure {
```

```
    private static final String SPROC_NAME = "AllTitlesAndGenres";

    public TitlesAndGenresStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR, new GenreMapper()));
        compile();
    }

    public Map execute() {
        // again, this sproc has no input parameters, so an empty Map is supplied...
        return super.execute(new HashMap());
    }
}
```

Notice how the overloaded variants of the `declareParameter(..)` method that have been used in the `TitlesAndGenresStoredProcedure` constructor are passed `RowMapper` implementation instances; this is a very convenient and powerful way to reuse existing functionality. (The code for the two `RowMapper` implementations is provided below in the interest of completeness.)

Firstly the `TitleMapper` class, which simply maps a `ResultSet` to a `Title` domain object for each row in the supplied `ResultSet`.

```
import com.foo.sprocs.domain.Title;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public final class TitleMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Title title = new Title();
        title.setId(rs.getLong("id"));
        title.setName(rs.getString("name"));
        return title;
    }
}
```

Secondly, the `GenreMapper` class, which again simply maps a `ResultSet` to a `Genre` domain object for each row in the supplied `ResultSet`.

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Genre;

public final class GenreMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

If one needs to pass parameters to a stored procedure (that is the stored procedure has been declared as having one or more input parameters in its definition in the RDBMS), one would code a strongly typed `execute(..)` method which would delegate to the superclass' (untyped) `execute(Map parameters)` (which has `protected` access); for example:

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
```

```
import java.util.HashMap;
import java.util.Map;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate";

    public TitlesAfterDateStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlParameter(CUTOFF_DATE_PARAM, Types.DATE);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR, new TitleMapper()));
        compile();
    }

    public Map execute(Date cutoffDate) {
        Map inputs = new HashMap();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

11.4.5. `SqlFunction`

The `SqlFunction` RDBMS operation class encapsulates an 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 the `StoredProcedure` or `SqlCall` classes for this type of processing).

`SqlFunction` is a concrete class, and there is typically no need to subclass it. 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 12. Object Relational Mapping (ORM) data access

12.1. Introduction

The Spring Framework provides integration with *Hibernate, JDO, Oracle TopLink, iBATIS SQL Maps* and *JPA*: in terms of resource management, DAO implementation support, and transaction strategies. For example for Hibernate, there is first-class support with lots of IoC convenience features, addressing many typical Hibernate integration issues. All of these support packages for O/R (Object Relational) mappers comply with Spring's generic transaction and DAO exception hierarchies. There are usually two integration styles: either using Spring's DAO 'templates' or coding DAOs against plain Hibernate/JDO/TopLink/etc APIs. In both cases, DAOs can be configured through Dependency Injection and participate in Spring's resource and transaction management.

Spring adds significant support when using the O/R mapping layer of your choice to create data access applications. First of all, you should know that once you started using Spring's support for O/R mapping, you don't have to go all the way. No matter to what extent, you're invited to review and leverage the Spring approach, before deciding to take the effort and risk of building a similar infrastructure in-house. Much of the O/R mapping support, no matter what technology you're using may be used in a library style, as everything is designed as a set of reusable JavaBeans. Usage inside a Spring IoC container does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside a Spring container.

Some of the benefits of using the Spring Framework to create your ORM DAOs include:

- *Ease of testing.* Spring's IoC approach makes it easy to swap the implementations and config locations of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, 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.

- *Common data access exceptions.* Spring can wrap exceptions from your O/R mapping tool of choice, converting them from proprietary (potentially checked) exceptions to a common runtime DataAccessException hierarchy. 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.

- *General resource management.* Spring application contexts can handle the location and configuration of Hibernate `SessionFactory` instances, JDBC `DataSource` instances, iBATIS SQL Maps configuration objects, and other related resources. This makes these values easy to manage and change. Spring offers efficient, easy and safe handling of persistence resources. For example: related code using Hibernate generally needs to use the same Hibernate `Session` for efficiency and proper transaction handling. Spring makes it easy to transparently create and bind a `Session` to the current thread, either by using an explicit 'template' wrapper class at the Java code level or by exposing a current `Session` through the Hibernate `SessionFactory` (for DAOs based on plain Hibernate API). Thus Spring solves many of the issues that repeatedly arise from typical Hibernate usage, for any transaction environment (local or JTA).

- *Integrated transaction management.* Spring allows you to wrap your O/R mapping 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/JDO related code being affected: for example, between local transactions and JTA, with the same full services (such as declarative transactions) available in both scenarios. As an additional benefit, JDBC-related code can fully integrate transactionally with the code you use to do O/R mapping. This is useful for data access that's not suitable for O/R mapping, such as batch processing or streaming of BLOBs, which still needs to share common transactions with ORM operations.

The PetClinic sample in the Spring distribution offers alternative DAO implementations and application context configurations for JDBC, Hibernate, Oracle TopLink, and JPA. PetClinic can therefore serve as working sample app that illustrates the use of Hibernate, TopLink and JPA in a Spring web application. It also leverages declarative transaction demarcation with different transaction strategies.

The JPetStore sample illustrates the use of iBATIS SQL Maps in a Spring environment. It also features two web tier versions: one based on Spring Web MVC, one based on Struts.

Beyond the samples shipped with Spring, there are a variety of Spring-based O/R mapping samples provided by specific vendors: for example, the JDO implementations JPOX (http://www.jpox.org/) and Kodo (http://www.bea.com/kodo/).

12.2. Hibernate

We will start with a coverage of Hibernate 3 in a Spring environment, using it to demonstrate the approach that Spring takes towards integrating O/R mappers. This section will cover many issues in detail and show different variations of DAO implementations and transaction demarcations. Most of these patterns can be directly translated to all other supported ORM tools. The following sections in this chapter will then cover the other ORM technologies, showing briefer examples there.

The following discussion focuses on Hibernate 3: this is the current major production ready version of Hibernate. Hibernate 2.x, which has been supported in Spring since its inception continues to be supported... it is just that the following examples all use the Hibernate 3 classes and configuration. All of this can (pretty much) be applied to Hibernate 2.x as-is, using the analogous Hibernate 2.x support package: `org.springframework.orm.hibernate`, mirroring `org.springframework.orm.hibernate3` with analogous support classes for Hibernate 2.x. Furthermore, all references to the `org.hibernate` package need to be replaced with `net.sf.hibernate`, following the root package change in Hibernate 3. Simply adapt the package names (as used in the examples) accordingly.

12.2.1. 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, namely IoC via templating; for example 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 service 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 services (that use the data access objects and transaction managers), web view resolvers, web controllers (that use the business services),and so on.

12.2.2. `SessionFactory` setup in a Spring container

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

  <bean id="mySessionFactory" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.HSQLDialect
      </value>
    </property>
  </bean>

</beans>
```

Note that switching from a local Jakarta Commons DBCP `BasicDataSource` to a JNDI-located `DataSource` (usually managed by an application server) is just a matter of configuration:

```
<beans>

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

</beans>
```

You can also access a JNDI-located `SessionFactory`, using Spring's `JndiObjectFactoryBean` to retrieve and expose it. However, that is typically not common outside of an EJB context.

12.2.3. The `HibernateTemplate`

The basic programming model for templating looks as follows, for methods that can be part of any custom data access object or business service. 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 IoC container - via a simple `setSessionFactory(..)` bean property setter. The following snippets show a DAO definition in a Spring container, referencing the above defined `SessionFactory`, and an example for a DAO method implementation.

```
<beans>

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

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

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

    public Collection loadProductsByCategory(String category) throws DataAccessException {
            return this.hibernateTemplate.find("from test.Product product where product.category=?", category);
    }
}
```

The `HibernateTemplate` class provides many methods that mirror the methods exposed on the Hibernate `Session` interface, in addition to a number of convenience methods such as the one shown above. If you need access to the `Session` to invoke methods that are not exposed on the `HibernateTemplate`, you can always drop down to a callback-based approach like so.

```
public class ProductDaoImpl implements ProductDao {

    private HibernateTemplate hibernateTemplate;

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

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return this.hibernateTemplate.execute(new HibernateCallback() {
            public Object doInHibernate(Session session) {
                Criteria criteria = session.createCriteria(Product.class);
                criteria.add(Expression.eq("category", category));
                criteria.setMaxResults(6);
                return criteria.list();
            }
        };
    }
}
```

A callback implementation effectively can be used for any Hibernate data access. `HibernateTemplate` will ensure that `Session` instances are properly opened and closed, and automatically participate 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 Collection loadProductsByCategory(String category) throws DataAccessException {
        return this.getHibernateTemplate().find(
            "from test.Product product where product.category=?", category);
    }
}
```

12.2.4. Implementing Spring-based DAOs without callbacks

As alternative to using Spring's `HibernateTemplate` to implement DAOs, data access code can also be written in a more traditional fashion, without wrapping the Hibernate access code in a callback, while still respecting and participating in Spring's generic `DataAccessException` hierarchy. The `HibernateDaoSupport` base class offers methods to access the current transactional `Session` and to convert exceptions in such a scenario; similar methods are also available as static helpers on the `SessionFactoryUtils` class. Note that such code will usually pass '`false`' as the value of the `getSession(..)` methods '`allowCreate`' argument, to enforce running within a transaction (which avoids the need to close the returned `Session`, as its lifecycle is managed by the transaction).

```
public class HibernateProductDao extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException, MyException {
        Session session = getSession(false);
        try {
            Query query = session.createQuery("from test.Product product where product.category=?");
            query.setString(0, category);
            List result = query.list();
            if (result == null) {
                throw new MyException("No search results.");
            }
            return result;
        }
        catch (HibernateException ex) {
            throw convertHibernateAccessException(ex);
        }
    }
}
```

The advantage of such direct Hibernate access code is that it allows *any* checked application exception to be thrown within the data access code; contrast this to the `HibernateTemplate` class which is restricted to throwing only unchecked exceptions within the callback. Note that you can often defer the corresponding checks and the throwing of application exceptions to after the callback, which still allows working with `HibernateTemplate`. In general, the `HibernateTemplate` class' convenience methods are simpler and more convenient for many scenarios.

12.2.5. Implementing DAOs based on plain Hibernate 3 API

Hibernate 3.0.1 introduced a feature called "contextual Sessions", where Hibernate itself manages one current `Session` per transaction. This is roughly equivalent to Spring's synchronization of one Hibernate `Session` per transaction. A corresponding DAO implementation looks like as follows, based on the plain Hibernate API:

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

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

    public Collection loadProductsByCategory(String category) {
        return this.sessionFactory.getCurrentSession()
                .createQuery("from test.Product product where product.category=?")
                .setParameter(0, category)
```

```
                .list();
    }
}
```

This style is very similar to what you will find in the Hibernate reference documentation and examples, except for holding the `SessionFactory` in an instance variable. We strongly recommend such an instance-based setup over the old-school `static HibernateUtil` class from Hibernate's CaveatEmptor sample application. (In general, do not keep any resources in `static` variables unless *absolutely* necessary.)

The above DAO follows the Dependency Injection pattern: it fits nicely into a Spring IoC container, just like it would if coded against Spring's `HibernateTemplate`. Of course, such a DAO can also be set up in plain Java (for example, in unit tests): simply instantiate it and call `setSessionFactory(..)` with the desired factory reference. As a Spring bean definition, it would look as follows:

```
<beans>

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

</beans>
```

The main advantage of this DAO style is that it depends on Hibernate API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and will no doubt feel more natural to Hibernate developers.

However, the DAO throws plain `HibernateException` (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on Hibernate's own exception hierarchy. Catching specific causes such as an optimistic locking failure is not possible without tieing the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly Hibernate-based and/or do not need any special exception treatment.

Fortunately, Spring's `LocalSessionFactoryBean` supports Hibernate's `SessionFactory.getCurrentSession()` method for any Spring transaction strategy, returning the current Spring-managed transactional `Session` even with `HibernateTransactionManager`. Of course, the standard behavior of that method remains: returning the current `Session` associated with the ongoing JTA transaction, if any (no matter whether driven by Spring's `JtaTransactionManager`, by EJB CMT, or by JTA).

In summary: DAOs can be implemented based on the plain Hibernate 3 API, while still being able to participate in Spring-managed transactions.

12.2.6. Programmatic transaction demarcation

Transactions can be demarcated in a higher level of the application, on top of such lower-level data access services spanning any number of operations. There are no restrictions on the implementation of the surrounding business service here as well, 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 service definition in a Spring application context, and an example for a business method implementation.

```
<beans>

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
```

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

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private TransactionTemplate transactionTemplate;
    private ProductDao productDao;

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

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

    public void increasePriceOfAllProductsInCategory(final String category) {
        this.transactionTemplate.execute(new TransactionCallbackWithoutResult() {

                public void doInTransactionWithoutResult(TransactionStatus status) {
                    List productsToChange = this.productDao.loadProductsByCategory(category);
                    // do the price increase...
                }
            }
        );
    }
}
```

12.2.7. Declarative transaction demarcation

Alternatively, one can use Spring's declarative transaction support, which essentially enables you to replace explicit transaction demarcation API calls in your Java code with an AOP transaction interceptor configured in a Spring container. This allows you to keep business services free of repetitive transaction demarcation code, and allows you to focus on adding business logic which is where the real value of your application lies. Furthermore, transaction semantics like propagation behavior and isolation level can be changed in a configuration file and do not affect the business service implementations.

```
<beans>

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <bean id="myProductService" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces" value="product.ProductService"/>
    <property name="target">
        <bean class="product.DefaultProductService">
            <property name="productDao" ref="myProductDao"/>
        </bean>
    </property>
    <property name="interceptorNames">
      <list>
        <value>myTxInterceptor</value> <!-- the transaction interceptor (configured elsewhere) -->
      </list>
    </property>
  </bean>

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

    public void setProductDao(ProductDao productDao) {
```

```
        this.productDao = productDao;
    }

    // notice the absence of transaction demarcation code in this method
    // Spring's declarative transaction infrastructure will be demarcating transactions on your behalf
    public void increasePriceOfAllProductsInCategory(final String category) {
        List productsToChange = this.productDao.loadProductsByCategory(category);
        // ...
    }
}
```

Spring's `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.

The following higher level approach to declarative transactions doesn't use the `ProxyFactoryBean`, and as such may be easier to use if you have a large number of service objects that you wish to make transactional.

Note

You are *strongly* encouraged to read the section entitled Section 9.5, “Declarative transaction management” if you have not done so already prior to continuing.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <!-- SessionFactory, DataSource, etc. omitted -->

  <bean id="myTxManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

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

</beans>
```

12.2.8. 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, such as 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 it's respective bean property. If all underlying JDBC data sources are transactional container ones, a business service 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="java:comp/env/jdbc/myds1"/>
  </bean>

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

  <bean id="mySessionFactory1" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource1"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.MySQLDialect
        hibernate.show_sql=true
      </value>
    </property>
  </bean>

  <bean id="mySessionFactory2" class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource2"/>
    <property name="mappingResources">
      <list>
        <value>inventory.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <value>
        hibernate.dialect=org.hibernate.dialect.OracleDialect
      </value>
    </property>
  </bean>

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

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

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

  <!-- this shows the Spring 1.x style of declarative transaction configuration -->
  <!-- it is totally supported, 100% legal in Spring 2.x, but see also above for the sleeker, Spring 2.0 style -
  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target">
      <bean class="product.ProductServiceImpl">
```

```
        <property name="productDao" ref="myProductDao"/>
        <property name="inventoryDao" ref="myInventoryDao"/>
      </bean>
    </property>
    <property name="transactionAttributes">
      <props>
        <prop key="increasePrice*">PROPAGATION_REQUIRED</prop>
        <prop key="someOtherBusinessMethod">PROPAGATION_REQUIRES_NEW</prop>
        <prop key="*">PROPAGATION_SUPPORTS,readOnly</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).

`HibernateTransactionManager` can export the JDBC `Connection` used by Hibernate to plain JDBC access code, for a specific `DataSource`. This allows for high-level transaction demarcation with mixed Hibernate/JDBC data access completely without JTA, as long as you are just accessing one database! `HibernateTransactionManager` will automatically expose the Hibernate transaction as JDBC transaction if the passed-in `SessionFactory` has been set up with a `DataSource` (through the "dataSource" property of the `LocalSessionFactoryBean` class). Alternatively, the `DataSource` that the transactions are supposed to be exposed for can also be specified explicitly, through the "dataSource" property of the `HibernateTransactionManager` class.

12.2.9. Container resources versus local resources

Spring's resource management allows for simple switching between a JNDI `SessionFactory` and a local one, without having to change a single line of application code. The decision as to 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. Deploying a `SessionFactory` through Hibernate's JCA connector provides the added value of participating in the J2EE server's management infrastructure, but does not add actual value beyond that.

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 as well. JTA does not just involve container dependencies in terms of JTA itself and of JNDI `DataSource` instances. For non-Spring JTA-driven Hibernate transactions, you have to use the Hibernate JCA connector, or extra Hibernate transaction code with the `TransactionManagerLookup` 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 for example WebLogic Express does not provide JCA. A Spring application 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 of the 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 really only adds value when used in conjunction with EJBs.

12.2.10. Spurious application server warnings when using Hibernate

In some JTA environments with very strict `XADataSource` implementations -- currently only some WebLogic and WebSphere versions -- when using Hibernate configured without any awareness of the JTA `PlatformTransactionManager` object for that environment, it is possible for spurious warning or exceptions to show up in the application server log. These warnings or exceptions will say something to the effect that the connection being accessed is no longer valid, or JDBC access is no longer valid, possibly because the transaction is no longer active. As an example, here is an actual exception from WebLogic:

```
java.sql.SQLException: The transaction is no longer active - status: 'Committed'.
   No further JDBC access is allowed within this transaction.
```

This warning is easy to resolve by simply making Hibernate aware of the JTA `PlatformTransactionManager` instance, to which it will also synchronize (along with Spring). This may be done in two ways:

- If in your application context you are already directly obtaining the JTA `PlatformTransactionManager` object (presumably from JNDI via `JndiObjectFactoryBean`) and feeding it for example to Spring's `JtaTransactionManager`, then the easiest way is to simply specify a reference to this as the value of `LocalSessionFactoryBean`'s *jtaTransactionManager* property. Spring will then make the object available to Hibernate.
- More likely you do not already have the JTA `PlatformTransactionManager` instance (since Spring's `JtaTransactionManager` can find it itself) so you need to instead configure Hibernate to also look it up directly. This is done by configuring an AppServer specific `TransactionManagerLookup` class in the Hibernate configuration, as described in the Hibernate manual.

It is not necessary to read any more for proper usage, but the full sequence of events with and without Hibernate being aware of the JTA `PlatformTransactionManager` will now be described.

When Hibernate is not configured with any awareness of the JTA `PlatformTransactionManager`, the sequence of events when a JTA transaction commits is as follows:

- JTA transaction commits
- Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so it is called back via an *afterCompletion* callback by the JTA transaction manager.
- Among other activities, this can trigger a callback by Spring to Hibernate, via Hibernate's `afterTransactionCompletion` callback (used to clear the Hibernate cache), followed by an explicit `close()` call on the Hibernate Session, which results in Hibernate trying to `close()` the JDBC Connection.
- In some environments, this `Connection.close()` call then triggers the warning or error, as the application server no longer considers the `Connection` usable at all, since the transaction has already been committed.

When Hibernate is configured with awareness of the JTA `PlatformTransactionManager`, the sequence of

events when a JTA transaction commits is instead as follows:

- JTA transaction is ready to commit
- Spring's `JtaTransactionManager` is synchronized to the JTA transaction, so it is called back via a *beforeCompletion* callback by the JTA transaction manager.
- Spring is aware that Hibernate itself is synchronized to the JTA transaction, and behaves differently than in the previous scenario. Assuming the Hibernate `Session` needs to be closed at all, Spring will close it now.
- JTA Transaction commits
- Hibernate is synchronized to the JTA transaction, so it is called back via an *afterCompletion* callback by the JTA transaction manager, and can properly clear its cache.

12.3. JDO

Spring supports the standard JDO 1.0/2.0 API as data access strategy, following the same style as the Hibernate support. The corresponding integration classes reside in the `org.springframework.orm.jdo` package.

12.3.1. `PersistenceManagerFactory` setup

Spring provides a `LocalPersistenceManagerFactoryBean` class that allows for defining a local JDO `PersistenceManagerFactory` within a Spring application context:

```
<beans>

  <bean id="myPmf" class="org.springframework.orm.jdo.LocalPersistenceManagerFactoryBean">
    <property name="configLocation" value="classpath:kodo.properties"/>
  </bean>

</beans>
```

Alternatively, a `PersistenceManagerFactory` can also be set up through direct instantiation of a `PersistenceManagerFactory` implementation class. A JDO `PersistenceManagerFactory` implementation class is supposed to follow the JavaBeans pattern, just like a JDBC `DataSource` implementation class, which is a natural fit for a Spring bean definition. This setup style usually supports a Spring-defined JDBC `DataSource`, passed into the "connectionFactory" property. For example, for the open source JDO implementation JPOX (http://www.jpox.org):

```
<beans>

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

 <bean id="myPmf" class="org.jpox.PersistenceManagerFactoryImpl" destroy-method="close">
   <property name="connectionFactory" ref="dataSource"/>
   <property name="nontransactionalRead" value="true"/>
 </bean>

</beans>
```

A JDO `PersistenceManagerFactory` can also be set up in the JNDI environment of a J2EE application server,

usually through the JCA connector provided by the particular JDO implementation. Spring's standard `JndiObjectFactoryBean` can be used to retrieve and expose such a `PersistenceManagerFactory`. However, outside an EJB context, there is often no compelling benefit in holding the `PersistenceManagerFactory` in JNDI: only choose such setup for a good reason. See "container resources versus local resources" in the Hibernate section for a discussion; the arguments there apply to JDO as well.

12.3.2. `JdoTemplate` and `JdoDaoSupport`

Each JDO-based DAO will then receive the `PersistenceManagerFactory` through dependency injection. Such a DAO could be coded against plain JDO API, working with the given `PersistenceManagerFactory`, but will usually rather be used with the Spring Framework's `JdoTemplate`:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private JdoTemplate jdoTemplate;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.jdoTemplate = new JdoTemplate(pmf);
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return (Collection) this.jdoTemplate.execute(new JdoCallback() {
            public Object doInJdo(PersistenceManager pm) throws JDOException {
                Query query = pm.newQuery(Product.class, "category = pCategory");
                query.declareParameters("String pCategory");
                List result = query.execute(category);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

A callback implementation can effectively be used for any JDO data access. `JdoTemplate` will ensure that `PersistenceManagers` are properly opened and closed, and automatically participate 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 such as a single `find`, `load`, `makePersistent`, or `delete` call, `JdoTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `JdoDaoSupport` base class that provides a `setPersistenceManagerFactory(..)` method for receiving a `PersistenceManagerFactory`, and `getPersistenceManagerFactory()` and `getJdoTemplate()` for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends JdoDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        return getJdoTemplate().find(
            Product.class, "category = pCategory", "String category", new Object[] {category});
    }
}
```

As alternative to working with Spring's `JdoTemplate`, you can also code Spring-based DAOs at the JDO API level, explicitly opening and closing a `PersistenceManager`. As elaborated in the corresponding Hibernate section, the main advantage of this approach is that your data access code is able to throw checked exceptions.

`JdoDaoSupport` offers a variety of support methods for this scenario, for fetching and releasing a transactional `PersistenceManager` as well as for converting exceptions.

12.3.3. Implementing DAOs based on the plain JDO API

DAOs can also be written against plain JDO API, without any Spring dependencies, directly using an injected `PersistenceManagerFactory`. A corresponding DAO implementation looks like as follows:

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        try {
            Query query = pm.newQuery(Product.class, "category = pCategory");
            query.declareParameters("String pCategory");
            return query.execute(category);
        }
        finally {
          pm.close();
        }
    }
}
```

As the above DAO still follows the Dependency Injection pattern, it still fits nicely into a Spring container, just like it would if coded against Spring's `JdoTemplate`:

```
<beans>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

The main issue with such DAOs is that they always get a new `PersistenceManager` from the factory. To still access a Spring-managed transactional `PersistenceManager`, consider defining a `TransactionAwarePersistenceManagerFactoryProxy` (as included in Spring) in front of your target `PersistenceManagerFactory`, passing the proxy into your DAOs.

```
<beans>

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

</beans>
```

Your data access code will then receive a transactional `PersistenceManager` (if any) from the `PersistenceManagerFactory.getPersistenceManager()` method that it calls. The latter method call goes through the proxy, which will first check for a current transactional `PersistenceManager` before getting a new one from the factory. `close()` calls on the `PersistenceManager` will be ignored in case of a transactional `PersistenceManager`.

If your data access code will always run within an active transaction (or at least within active transaction synchronization), it is safe to omit the `PersistenceManager.close()` call and thus the entire `finally` block, which you might prefer to keep your DAO implementations concise:

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(String category) {
        PersistenceManager pm = this.persistenceManagerFactory.getPersistenceManager();
        Query query = pm.newQuery(Product.class, "category = pCategory");
        query.declareParameters("String pCategory");
        return query.execute(category);
    }
}
```

With such DAOs that rely on active transactions, it is recommended to enforce active transactions through turning `TransactionAwarePersistenceManagerFactoryProxy`'s "allowCreate" flag off:

```
<beans>

  <bean id="myPmfProxy"
      class="org.springframework.orm.jdo.TransactionAwarePersistenceManagerFactoryProxy">
    <property name="targetPersistenceManagerFactory" ref="myPmf"/>
    <property name="allowCreate" value="false"/>
  </bean>

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmfProxy"/>
  </bean>

</beans>
```

The main advantage of this DAO style is that it depends on JDO API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and might feel more natural to JDO developers.

However, the DAO throws plain `JDOException` (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on JDO's own exception structure. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly JDO-based and/or do not need any special exception treatment.

In summary: DAOs can be implemented based on plain JDO API, while still being able to participate in Spring-managed transactions. This might in particular appeal to people already familiar with JDO, feeling more natural to them. However, such DAOs will throw plain `JDOException`; conversion to Spring's `DataAccessException` would have to happen explicitly (if desired).

12.3.4. Transaction management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
```

```
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <bean id="myTxManager" class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

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

  <tx:advice id="txAdvice" transaction-manager="txManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

</beans>
```

Note that JDO requires an active transaction when modifying a persistent object. There is no concept like a non-transactional flush in JDO, in contrast to Hibernate. For this reason, the chosen JDO implementation needs to be set up for a specific environment: in particular, it needs to be explicitly set up for JTA synchronization, to detect an active JTA transaction itself. This is not necessary for local transactions as performed by Spring's `JdoTransactionManager`, but it is necessary for participating in JTA transactions (whether driven by Spring's `JtaTransactionManager` or by EJB CMT / plain JTA).

`JdoTransactionManager` is capable of exposing a JDO transaction to JDBC access code that accesses the same JDBC `DataSource`, provided that the registered `JdoDialect` supports retrieval of the underlying JDBC `Connection`. This is by default the case for JDBC-based JDO 2.0 implementations; for JDO 1.0 implementations, a custom `JdoDialect` needs to be used. See next section for details on the `JdoDialect` mechanism.

12.3.5. `JdoDialect`

As an advanced feature, both `JdoTemplate` and `interfacename` support a custom `JdoDialect`, to be passed into the "jdoDialect" bean property. In such a scenario, the DAOs won't receive a `PersistenceManagerFactory` reference but rather a full `JdoTemplate` instance instead (for example, passed into `JdoDaoSupport`'s "jdoTemplate" property). A `JdoDialect` implementation can enable some advanced features supported by Spring, usually in a vendor-specific manner:

- applying specific transaction semantics (such as custom isolation level or transaction timeout)
- retrieving the transactional JDBC `Connection` (for exposure to JDBC-based DAOs)
- applying query timeouts (automatically calculated from Spring-managed transaction timeout)
- eagerly flushing a `PersistenceManager` (to make transactional changes visible to JDBC-based data access code)
- advanced translation of `JDOExceptions` to Spring `DataAccessExceptions`

This is particularly valuable for JDO 1.0 implementations, where none of those features are covered by the standard API. On JDO 2.0, most of those features are supported in a standard manner: Hence, Spring's `DefaultJdoDialect` uses the corresponding JDO 2.0 API methods by default (as of Spring 1.2). For special transaction semantics and for advanced translation of exception, it is still valuable to derive vendor-specific `JdoDialect` subclasses.

See the `JdoDialect` Javadoc for more details on its operations and how they are used within Spring's JDO support.

12.4. Oracle TopLink

Since Spring 1.2, Spring supports Oracle TopLink (http://www.oracle.com/technology/products/ias/toplink) as data access strategy, following the same style as the Hibernate support. Both TopLink 9.0.4 (the production version as of Spring 1.2) and 10.1.3 (still in beta as of Spring 1.2) are supported. The corresponding integration classes reside in the `org.springframework.orm.toplink` package.

Spring's TopLink support has been co-developed with the Oracle TopLink team. Many thanks to the TopLink team, in particular to Jim Clark who helped to clarify details in all areas!

12.4.1. `SessionFactory` abstraction

TopLink itself does not ship with a SessionFactory abstraction. Instead, multi-threaded access is based on the concept of a central `ServerSession`, which in turn is able to spawn `ClientSession` instances for single-threaded usage. For flexible setup options, Spring defines a `SessionFactory` abstraction for TopLink, enabling to switch between different `Session` creation strategies.

As a one-stop shop, Spring provides a `LocalSessionFactoryBean` class that allows for defining a TopLink `SessionFactory` with bean-style configuration. It needs to be configured with the location of the TopLink session configuration file, and usually also receives a Spring-managed JDBC `DataSource` to use.

```
<beans>

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

  <bean id="mySessionFactory" class="org.springframework.orm.toplink.LocalSessionFactoryBean">
    <property name="configLocation" value="toplink-sessions.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

</beans>
```

```
<toplink-configuration>

  <session>
    <name>Session</name>
    <project-xml>toplink-mappings.xml</project-xml>
    <session-type>
      <server-session/>
    </session-type>
    <enable-logging>true</enable-logging>
    <logging-options/>
  </session>

</toplink-configuration>
```

Usually, `LocalSessionFactoryBean` will hold a multi-threaded TopLink `ServerSession` underneath and create appropriate client `Sessions` for it: either a plain `Session` (typical), a managed `ClientSession`, or a transaction-aware `Session` (the latter are mainly used internally by Spring's TopLink support). It might also hold a single-threaded TopLink `DatabaseSession`; this is rather unusual, though.

12.4.2. `TopLinkTemplate` and `TopLinkDaoSupport`

Each TopLink-based DAO will then receive the `SessionFactory` through dependency injection, i.e. through a bean property setter or through a constructor argument. Such a DAO could be coded against plain TopLink API, fetching a `Session` from the given `SessionFactory`, but will usually rather be used with Spring's `TopLinkTemplate`:

```
<beans>

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

</beans>
```

```
public class TopLinkProductDao implements ProductDao {

    private TopLinkTemplate tlTemplate;

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

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return (Collection) this.tlTemplate.execute(new TopLinkCallback() {
            public Object doInTopLink(Session session) throws TopLinkException {
                ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
                findOwnersQuery.addArgument("Category");
                ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
                findOwnersQuery.setSelectionCriteria(
                    builder.get("category").like(builder.getParameter("Category")));

                Vector args = new Vector();
                args.add(category);
                List result = session.executeQuery(findOwnersQuery, args);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

A callback implementation can effectively be used for any TopLink data access. `TopLinkTemplate` will ensure that `Sessions` are properly opened and closed, and automatically participate 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 such as a single `executeQuery`, `readAll`, `readById`, or `merge` call, `JdoTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `TopLinkDaoSupport` base class that provides a `setSessionFactory(..)` method for receiving a `SessionFactory`, and `getSessionFactory()` and `getTopLinkTemplate()` for use by subclasses. In combination, this allows for simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends TopLinkDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));
```

```
        return getTopLinkTemplate().executeQuery(findOwnersQuery, new Object[] {category});
    }
}
```

Side note: TopLink query objects are thread-safe and can be cached within the DAO, i.e. created on startup and kept in instance variables.

As alternative to working with Spring's `TopLinkTemplate`, you can also code your TopLink data access based on the raw TopLink API, explicitly opening and closing a `Session`. As elaborated in the corresponding Hibernate section, the main advantage of this approach is that your data access code is able to throw checked exceptions. `TopLinkDaoSupport` offers a variety of support methods for this scenario, for fetching and releasing a transactional `Session` as well as for converting exceptions.

12.4.3. Implementing DAOs based on plain TopLink API

DAOs can also be written against plain TopLink API, without any Spring dependencies, directly using an injected TopLink `Session`. The latter will usually be based on a `SessionFactory` defined by a `LocalSessionFactoryBean`, exposed for bean references of type `Session` through Spring's `TransactionAwareSessionAdapter`.

The `getActiveSession()` method defined on TopLink's `Session` interface will return the current transactional `Session` in such a scenario. If there is no active transaction, it will return the shared TopLink `ServerSession` as-is, which is only supposed to be used directly for read-only access. There is also an analogous `getActiveUnitOfWork()` method, returning the TopLink `UnitOfWork` associated with the current transaction, if any (returning `null` else).

A corresponding DAO implementation looks like as follows:

```
public class ProductDaoImpl implements ProductDao {

    private Session session;

    public void setSession(Session session) {
        this.session = session;
    }

    public Collection loadProductsByCategory(String category) {
        ReadAllQuery findOwnersQuery = new ReadAllQuery(Product.class);
        findOwnersQuery.addArgument("Category");
        ExpressionBuilder builder = this.findOwnersQuery.getExpressionBuilder();
        findOwnersQuery.setSelectionCriteria(
            builder.get("category").like(builder.getParameter("Category")));

        Vector args = new Vector();
        args.add(category);
        return session.getActiveSession().executeQuery(findOwnersQuery, args);
    }
}
```

As the above DAO still follows the Dependency Injection pattern, it still fits nicely into a Spring application context, analogous to like it would if coded against Spring's `TopLinkTemplate`. Spring's `TransactionAwareSessionAdapter` is used to expose a bean reference of type `Session`, to be passed into the DAO:

```
<beans>

  <bean id="mySessionAdapter"
      class="org.springframework.orm.toplink.support.TransactionAwareSessionAdapter">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>
```

```
  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="session" ref="mySessionAdapter"/>
  </bean>

</beans>
```

The main advantage of this DAO style is that it depends on TopLink API only; no import of any Spring class is required. This is of course appealing from a non-invasiveness perspective, and might feel more natural to TopLink developers.

However, the DAO throws plain `TopLinkException` (which is unchecked, so does not have to be declared or caught), which means that callers can only treat exceptions as generally fatal - unless they want to depend on TopLink's own exception structure. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly TopLink-based and/or do not need any special exception treatment.

A further disadvantage of that DAO style is that TopLink's standard `getActiveSession()` feature just works within JTA transactions. It does *not* work with any other transaction strategy out-of-the-box, in particular not with local TopLink transactions.

Fortunately, Spring's `TransactionAwareSessionAdapter` exposes a corresponding proxy for the TopLink `ServerSession` which supports TopLink's `Session.getActiveSession()` and `Session.getActiveUnitOfWork()` methods for any Spring transaction strategy, returning the current Spring-managed transactional `Session` even with `TopLinkTransactionManager`. Of course, the standard behavior of that method remains: returning the current `Session` associated with the ongoing JTA transaction, if any (no matter whether driven by Spring's `JtaTransactionManager`, by EJB CMT, or by plain JTA).

In summary: DAOs can be implemented based on plain TopLink API, while still being able to participate in Spring-managed transactions. This might in particular appeal to people already familiar with TopLink, feeling more natural to them. However, such DAOs will throw plain `TopLinkException`; conversion to Spring's `DataAccessException` would have to happen explicitly (if desired).

12.4.4. Transaction management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans
        xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:aop="http://www.springframework.org/schema/aop"
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation="
   http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
   http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
   http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <bean id="myTxManager" class="org.springframework.orm.toplink.TopLinkTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

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

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>
```

```
  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>

</beans>
```

Note that TopLink requires an active `UnitOfWork` for modifying a persistent object. (You should never modify objects returned by a plain TopLink `Session` - those are usually read-only objects, directly taken from the second-level cache!) There is no concept like a non-transactional flush in TopLink, in contrast to Hibernate. For this reason, TopLink needs to be set up for a specific environment: in particular, it needs to be explicitly set up for JTA synchronization, to detect an active JTA transaction itself and expose a corresponding active `Session` and `UnitOfWork`. This is not necessary for local transactions as performed by Spring's `TopLinkTransactionManager`, but it is necessary for participating in JTA transactions (whether driven by Spring's `JtaTransactionManager` or by EJB CMT / plain JTA).

Within your TopLink-based DAO code, use the `Session.getActiveUnitOfWork()` method to access the current `UnitOfWork` and perform write operations through it. This will only work within an active transaction (both within Spring-managed transactions and plain JTA transactions). For special needs, you can also acquire separate `UnitOfWork` instances that won't participate in the current transaction; this is hardly needed, though.

`TopLinkTransactionManager` is capable of exposing a TopLink transaction to JDBC access code that accesses the same JDBC `DataSource`, provided that TopLink works with JDBC in the backend and is thus able to expose the underlying JDBC `Connection`. The `DataSource` to expose the transactions for needs to be specified explicitly; it won't be autodetected.

12.5. iBATIS SQL Maps

The iBATIS support in the Spring Framework much resembles the JDBC / Hibernate support in that it supports the same template style programming and just as with JDBC or Hibernate, the iBATIS support works with Spring's exception hierarchy and let's you enjoy the all IoC features Spring has.

Transaction management can be handled through Spring's standard facilities. There are no special transaction strategies for iBATIS, as there is no special transactional resource involved other than a JDBC `Connection`. Hence, Spring's standard JDBC `DataSourceTransactionManager` or `JtaTransactionManager` are perfectly sufficient.

Note

Spring does actually support both iBatis 1.x and 2.x. However, only support for iBatis 2.x is actually shipped with the core Spring distribution. The iBatis 1.x support classes were moved to the Spring Modules project as of Spring 2.0, and you are directed there for documentation.

12.5.1. Setting up the `SqlMapClient`

If we want to map the previous Account class with iBATIS 2.x we need to create the following SQL map `'Account.xml'`:

```
<sqlMap namespace="Account">

  <resultMap id="result" class="examples.Account">
```

```
    <result property="name" column="NAME" columnIndex="1"/>
    <result property="email" column="EMAIL" columnIndex="2"/>
  </resultMap>

  <select id="getAccountByEmail" resultMap="result">
    select ACCOUNT.NAME, ACCOUNT.EMAIL
    from ACCOUNT
    where ACCOUNT.EMAIL = #value#
  </select>

  <insert id="insertAccount">
    insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
  </insert>

</sqlMap>
```

The configuration file for iBATIS 2 looks like this:

```
<sqlMapConfig>

  <sqlMap resource="example/Account.xml"/>

</sqlMapConfig>
```

Remember that iBATIS loads resources from the class path, so be sure to add the `'Account.xml'` file to the class path.

We can use the `SqlMapClientFactoryBean` in the Spring container. Note that with iBATIS SQL Maps 2.x, the JDBC `DataSource` is usually specified on the `SqlMapClientFactoryBean`, which enables lazy loading.

```
<beans>

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

  <bean id="sqlMapClient" class="org.springframework.orm.ibatis.SqlMapClientFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
    <property name="dataSource" ref="dataSource"/>
  </bean>

</beans>
```

12.5.2. Using `SqlMapClientTemplate` and `SqlMapClientDaoSupport`

The `SqlMapClientDaoSupport` class offers a supporting class similar to the `SqlMapDaoSupport`. We extend it to implement our DAO:

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        return (Account) getSqlMapClientTemplate().queryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().update("insertAccount", account);
    }
}
```

In the DAO, we use the pre-configured `SqlMapClientTemplate` to execute the queries, after setting up the `SqlMapAccountDao` in the application context and wiring it with our `SqlMapClient` instance:

```
<beans>

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

Note that a `SqlMapTemplate` instance could also be created manually, passing in the `SqlMapClient` as constructor argument. The `SqlMapClientDaoSupport` base class simply pre-initializes a `SqlMapClientTemplate` instance for us.

The `SqlMapClientTemplate` also offers a generic `execute` method, taking a custom `SqlMapClientCallback` implementation as argument. This can, for example, be used for batching:

```
public class SqlMapAccountDao extends SqlMapClientDaoSupport implements AccountDao {

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapClientTemplate().execute(new SqlMapClientCallback() {
            public Object doInSqlMapClient(SqlMapExecutor executor) throws SQLException {
                executor.startBatch();
                executor.update("insertAccount", account);
                executor.update("insertAddress", account.getAddress());
                executor.executeBatch();
            }
        });
    }
}
```

In general, any combination of operations offered by the native `SqlMapExecutor` API can be used in such a callback. Any `SQLException` thrown will automatically get converted to Spring's generic `DataAccessException` hierarchy.

12.5.3. Implementing DAOs based on plain iBATIS API

DAOs can also be written against plain iBATIS API, without any Spring dependencies, directly using an injected `SqlMapClient`. A corresponding DAO implementation looks like as follows:

```
public class SqlMapAccountDao implements AccountDao {

    private SqlMapClient sqlMapClient;

    public void setSqlMapClient(SqlMapClient sqlMapClient) {
        this.sqlMapClient = sqlMapClient;
    }

    public Account getAccount(String email) {
        try {
            return (Account) this.sqlMapClient.queryForObject("getAccountByEmail", email);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }

    public void insertAccount(Account account) throws DataAccessException {
        try {
            this.sqlMapClient.update("insertAccount", account);
        }
        catch (SQLException ex) {
            throw new MyDaoException(ex);
        }
    }
}
```

In such a scenario, the `SQLException` thrown by the iBATIS API needs to be handled in a custom fashion: usually, wrapping it in your own application-specific DAO exception. Wiring in the application context would still look like before, due to the fact that the plain iBATIS-based DAO still follows the Dependency Injection pattern:

```
<beans>

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="sqlMapClient" ref="sqlMapClient"/>
  </bean>

</beans>
```

12.6. JPA

Spring JPA (available under the `org.springframework.orm.jpa` package) offers comprehensive support for the Java Persistence API in a similar manner to the integration with Hibernate or JDO, while being aware of the underlying implementation in order to provide additional features.

12.6.1. JPA setup in a Spring environment

Spring JPA offers three ways of setting up JPA `EntityManagerFactory`:

12.6.1.1. `LocalEntityManagerFactoryBean`

The `LocalEntityManagerFactoryBean` creates an `EntityManagerFactory` suitable for environments which solely use JPA for data access. The factory bean will use the JPA `PersistenceProvider` autodetection mechanism (according to JPA's Java SE bootstrapping) and, in most cases, requires only the persistence unit name to be specified:

```
<beans>

   <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
      <property name="persistenceUnitName" value="myPersistenceUnit"/>
   </bean>

</beans>
```

This is the simplest but also most limited form of JPA deployment. There is no way to link to an existing JDBC `DataSource` and no support for global transactions, for example. Furthermore, weaving (byte-code transformation) of persistent classes is provider-specific, often requiring a specific JVM agent to specified on startup. All in all, this option is only really sufficient for standalone applications and test environments (which is exactly what the JPA specification designed it for).

Only use this option in simple deployment environments like standalone applications and integration tests.

12.6.1.2. `Obtaining an EntityManagerFactory from JNDI`

Obtaining an `EntityManagerFactory` from JNDI (for example in a Java EE 5 environment), is just a matter of changing the XML configuration:

```
<beans>

    <jee:jndi-lookup id="entityManagerFactory" jndi-name="persistence/myPersistenceUnit"/>

</beans>
```

This assumes standard Java EE 5 bootstrapping, with the Java EE server autodetecting persistence units (i.e. `META-INF/persistence.xml` files in application jars) and `persistence-unit-ref` entries in the Java EE deployment descriptor (e.g. `web.xml`) defining environment naming context locations for those persistence units.

In such a scenario, the entire persistence unit deployment, including the weaving (byte-code transformation) of persistent classes, is up to the Java EE server. The JDBC `DataSource` is defined through a JNDI location in the `META-INF/persistence.xml` file; EntityManager transactions are integrated with the server's JTA subsystem. Spring merely uses the obtained `EntityManagerFactory`, passing it on to application objects via dependency injection, and managing transactions for it (typically through `JtaTransactionManager`).

Note that, in case of multiple persistence units used in the same application, the bean names of such a JNDI-retrieved persistence units should match the persistence unit names that the application uses to refer to them (e.g. in `@PersistenceUnit` and `@PersistenceContext` annotations).

Use this option when deploying to a Java EE 5 server. Check your server's documentation on how to deploy a custom JPA provider into your server, allowing for a different provider than the server's default.

12.6.1.3. `LocalContainerEntityManagerFactoryBean`

The `LocalContainerEntityManagerFactoryBean` gives full control over `EntityManagerFactory` configuration and is appropriate for environments where fine-grained customization is required. The `LocalContainerEntityManagerFactoryBean` will create a `PersistenceUnitInfo` based on the `persistence.xml` file, the supplied `dataSourceLookup` strategy and the specified `loadTimeWeaver`. It is thus possible to work with custom DataSources outside of JNDI and to control the weaving process.

```
<beans>

 <bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="dataSource" ref="someDataSource"/>
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
  </property>
 </bean>

</beans>
```

This is the most powerful JPA setup option, allowing for flexible local configuration within the application. It supports links to an existing JDBC `DataSource`, supports both local and global transactions, etc. However, it also imposes requirements onto the runtime environment, such as the availability of a weaving-capable ClassLoader if the persistence provider demands byte-code transformation.

Note that this option may conflict with the built-in JPA capabilities of a Java EE 5 server. So when running in a full Java EE 5 environment, consider obtaining your `EntityManagerFactory` from JNDI. Alternatively, specify a custom "persistenceXmlLocation" on your `LocalContainerEntityManagerFactoryBean` definition, e.g. "META-INF/my-persistence.xml", and only include a descriptor with that name in your application jar files. Since the Java EE 5 server will only look for default `META-INF/persistence.xml` files, it will ignore such custom persistence units and hence avoid conflicts with a Spring-driven JPA setup upfront. (This applies to Resin 3.1, for example.)

Use this option for full JPA capabilities in a Spring-based application environment. This includes web containers such as Tomcat as well as standalone applications and integration tests with sophisticated persistence requirements.

When is load time weaving required?

> Not all JPA providers impose the need of a JVM agent (Hibernate being an example). If your provider does not require an agent or you have other alternatives (for example applying enhancements at build time through a custom compiler or an ant task) the load time weaver **should not** be used.

The `LoadTimeWeaver` interface is a Spring-provided class that allows JPA `ClassTransformer` instances to be plugged in a specific manner depending on the environment (web container/application server). Hooking `ClassTransformers` through a JDK 5.0 agent is typically not efficient - the agents work against the *entire virtual machine* and inspect *every* class that is loaded - something that is typically undesirable in a production server enviroment.

Spring provides a number of `LoadTimeWeaver` implementations for various environments, allowing `ClassTransformer` instances to be applied only *per ClassLoader* and not per VM.

12.6.1.3.1. Tomcat load-time weaving setup (5.0+)

Jakarta Tomcat's default ClassLoader does not support class transformation but allows custom ClassLoaders to be used. Spring offers the `TomcatInstrumentableClassLoader` (inside the `org.springframework.instrument.classloading.tomcat` package) which extends the Tomcat ClassLoader (`WebappClassLoader`) and allows JPA `ClassTransformer` instances to 'enhance' all classes loaded by it. In short, JPA transformers will be applied only inside a specific web application (which uses the `TomcatInstrumentableClassLoader`).

In order to use the custom ClassLoader on:

1. Copy `spring-tomcat-weaver.jar` into *$CATALINA_HOME*/server/lib (where *$CATALINA_HOME* represents the root of the Tomcat installation).

2. Instruct Tomcat to use the custom ClassLoader (instead of the default one) by editing the web application context file:

```
<Context path="/myWebApp" docBase="/my/webApp/location">
   <Loader loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader" />
</Context>
```

   Tomcat 5.0.x and 5.5.x series support several context locations: server configuration file (*$CATALINA_HOME/conf/server.xml*), the default context configuration (*$CATALINA_HOME/conf/context.xml*) that affects all deployed web applications and per-webapp configurations, deployed on the server *($CATALINA_HOME/conf/[enginename]/[hostname]/my-webapp-context.xml*) side or along with the webapp (*your-webapp.war/META-INF/context.xml*). For efficiency, inside the web-app configuration style is recommended since only applications which use JPA will use the custom ClassLoader. See the Tomcat 5.x documentation for more details about available context locations.

   Note that versions prior to 5.5.20 contained a bug in the XML configuration parsing preventing usage of `Loader` tag inside *server.xml* (no matter if a ClassLoader is specified or not (be it the official or a custom one). See Tomcat's bugzilla for more details.

   If you are using Tomcat 5.5.20+ you can set *useSystemClassLoaderAsParent* to `false` to fix the problem:

```
<Context path="/myWebApp" docBase="/my/webApp/location">
    <Loader loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"
            useSystemClassLoaderAsParent="false"/>
</Context>
```

1. Copy `spring-tomcat-weaver.jar` into *$CATALINA_HOME*/lib (where *$CATALINA_HOME* represents the root of the Tomcat installation).

2. Instruct Tomcat to use the custom ClassLoader (instead of the default one) by editing the web application context file:

```
<Context path="/myWebApp" docBase="/my/webApp/location">
    <Loader loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"/>
</Context>
```

   Tomcat 6.0.x (similar to 5.0.x/5.5.x) series support several context locations: server configuration file (*$CATALINA_HOME/conf/server.xml*), the default context configuration (*$CATALINA_HOME/conf/context.xml*) that affects all deployed web applications and per-webapp configurations, deployed on the server *($CATALINA_HOME/conf/[enginename]/[hostname]/my-webapp-context.xml*) side or along with the webapp (*your-webapp.war/META-INF/context.xml*). For efficiency, inside the web-app configuration style is recommended since only applications which use JPA will use the custom ClassLoader. See the Tomcat 5.x documentation for more details about available context locations.

- Tomcat 5.0.x/5.5.x

- Tomcat 6.0.x

The last step required on all Tomcat versions, is to use the appropriate the `LoadTimeWeaver` when configuring `LocalContainerEntityManagerFactoryBean`:

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
  </property>
</bean>
```

Using this technique, JPA applications relying on instrumentation, can run in Tomcat without the need of an agent. This is important especially when hosting applications which rely on different JPA implementations since the JPA transformers are applied only at ClassLoader level and thus, are isolated from each other.

Note

If TopLink is being used a JPA provider under Tomcat, please place the toplink-essentials jar under *$CATALINA_HOME*/shared/lib folder instead of your war.

12.6.1.3.2. OC4J load-time weaving setup (10.1.3.1+)

As Oracle's OC4J ClassLoader has native bytecode transformation support, switching from an JDK agent to a `LoadTimeWeaver` can be done just through the application Spring configuration:

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.oc4j.OC4JLoadTimeWeaver"/>
  </property>
</bean>
```

12.6.1.3.3. GlassFish load-time weaving setup

GlassFish application server provides out of the box, an instrumentation cable ClassLoader. Spring supports it through `GlassFishLoadTimeWeaver`:

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.glassfish.GlassFishLoadTimeWeaver"/>
  </property>
</bean>
```

12.6.1.3.4. Resin load-time weaving setup (3.1+)

Caucho Resin 3.1 sesies ClassLoader provides native bytecode capabilities which can be used by Spring through `ReflectiveLoadTimeWeaver`:

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.ReflectiveLoadTimeWeaver"/>
  </property>
</bean>
```

12.6.1.3.5. General `LoadTimeWeaver`

For environments where class instrumentation is required but are not supported by the existing LoadTimeWeaver implementations, a JDK agent can be the only solution. For such cases, Spring provides `InstrumentationLoadTimeWeaver` which requires a Spring-specific (but very general) VM agent (`spring-agent.jar`):

```
<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
  <property name="loadTimeWeaver">
    <bean class="org.springframework.instrument.classloading.InstrumentationLoadTimeWeaver"/>
  </property>
</bean>
```

Note that the virtual machine has to be started with the Spring agent, by supplying the following JVM options:

```
-javaagent:/path/to/spring-agent.jar
```

12.6.1.4. Dealing with multiple persistence units

For applications that rely on multiple persistence units locations (stored in various jars in the classpath for example), Spring offers the `PersistenceUnitManager` to act as a central repository and avoid the (potentially expensive) persistence units discovery process. The default implementation allows multiple locations to be specified (by default, the classpath is searched for `'META-INF/persistence.xml'` files) which are parsed and later on retrieved through the persistence unit name:

```
<bean id="persistenceUnitManager" class="org.springframework.orm.jpa.persistenceunit.DefaultPersistenceUnitManag
  <property name="persistenceXmlLocation">
    <list>
     <value>org/springframework/orm/jpa/domain/persistence-multi.xml</value>
     <value>classpath:/my/package/**/custom-persistence.xml</value>
     <value>classpath*:META-INF/persistence.xml</value>
    </list>
  </property>
  <property>
   <map>
    <entry key="localDataSource" value-ref="local-db"/>
    <entry key="remoteDataSource" value-ref="remote-db"/>
   </map>
  </property>
  <!-- if no datasource is specified, use this one -->
  <property name="defaultDataSource" ref="remoteDataSource"/>
```

```
</bean>

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
 <property name="persistenceUnitManager" ref="persistenceUnitManager"/>
</bean>
```

Note that the default implementation allows customization of the persistence unit infos before feeding them to the JPA provider declaratively through its properties (which affect *all* housed units) or programmatically, through the `PersistenceUnitPostProcessor` (which allow persistence unit selection). If no persistenceUnitManager is specified, one will be created and used internally by `LocalContainerEntityManagerFactoryBean`.

12.6.2. `JpaTemplate` and `JpaDaoSupport`

Each JPA-based DAO will then receive a `EntityManagerFactory` via dependency injection. Such a DAO can be coded against plain JPA and work with the given `EntityManagerFactory` or through Spring's `JpaTemplate`:

```
<beans>

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

</beans>
```

```
public class JpaProductDao implements ProductDao {

    private JpaTemplate jpaTemplate;

    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.jpaTemplate = new JpaTemplate(emf);
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        return (Collection) this.jpaTemplate.execute(new JpaCallback() {
            public Object doInJpa(EntityManager em) throws PersistenceException {
                Query query = em.createQuery("from Product as p where p.category = :category");
                query.setParameter("category", category);
                List result = query.getResultList();
                // do some further processing with the result list
                return result;
            }
        });
    }
}
```

The `JpaCallback` implementation allows any type of JPA data access. The `JpaTemplate` will ensure that `EntityManagers` are properly opened and closed and automatically participate in transactions. Moreover, the `JpaTemplate` properly handles exceptions, making sure resources are cleaned up and the appropriate transactions rolled back. The template instances are thread-safe and reusable and they can be kept as instance variable of the enclosing class. Note that `JpaTemplate` offers single-step actions such as find, load, merge, etc along with alternative convenience methods that can replace one line callback implementations.

Furthermore, Spring provides a convenient `JpaDaoSupport` base class that provides the `get/setEntityManagerFactory` and `getJpaTemplate()` to be used by subclasses:

```
public class ProductDaoImpl extends JpaDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        Map<String, String> params = new HashMap<String, String>();
        params.put("category", category);
        return getJpaTemplate().findByNamedParams("from Product as p where p.category = :category", params);
    }
```

```
}
```

Besides working with Spring's `JpaTemplate`, one can also code Spring-based DAOs against the JPA, doing one's own explicit `EntityManager` handling. As also elaborated in the corresponding Hibernate section, the main advantage of this approach is that your data access code is able to throw checked exceptions. `JpaDaoSupport` offers a variety of support methods for this scenario, for retrieving and releasing a transaction `EntityManager`, as well as for converting exceptions.

JpaTemplate mainly exists as a sibling of JdoTemplate and HibernateTemplate, offering the same style for people used to it. For newly started projects, consider adopting the native JPA style of coding data access objects instead, based on a "shared EntityManager" reference obtained through the JPA `@PersistenceContext` annotation (using Spring's `PersistenceAnnotationBeanPostProcessor`; see below for details.)

12.6.3. Implementing DAOs based on plain JPA

Note

While `EntityManagerFactory` instances are thread-safe, `EntityManager` instances are not. The injected JPA `EntityManager` behave just like an `EntityManager` fetched from an application server's JNDI environment, as defined by the JPA specification. It will delegate all calls to the current transactional `EntityManager`, if any; else, it will fall back to a newly created `EntityManager` per operation, making it thread-safe.

It is possible to write code against the plain JPA without using any Spring dependencies, using an injected `EntityManagerFactory` or `EntityManager`. Note that Spring can understand `@PersistenceUnit` and `@PersistenceContext` annotations both at field and method level if a `PersistenceAnnotationBeanPostProcessor` is enabled. A corresponding DAO implementation might look like this:

```
public class ProductDaoImpl implements ProductDao {

    private EntityManagerFactory emf;

    @PersistenceUnit
    public void setEntityManagerFactory(EntityManagerFactory emf) {
        this.emf = emf;
    }

    public Collection loadProductsByCategory(String category) {
        EntityManager em = this.emf.createEntityManager();
        try {
             Query query = em.createQuery("from Product as p where p.category = ?1");
             query.setParameter(1, category);
             return query.getResultList();
        }
        finally {
            if (em != null) {
                em.close();
            }
        }
    }
}
```

The DAO above has no dependency on Spring and still fits nicely into a Spring application context, just like it would if coded against Spring's `JpaTemplate`. Moreover, the DAO takes advantage of annotations to require the injection of the default `EntityManagerFactory`:

```
<beans>

  <!-- bean post-processor for JPA annotations -->
  <bean class="org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

The main issue with such a DAO is that it always creates a new `EntityManager` via the factory. This can be easily overcome by requesting a transactional `EntityManager` (also called "shared EntityManager", since it is a shared, thread-safe proxy for the actual transactional EntityManager) to be injected instead of the factory:

```
public class ProductDaoImpl implements ProductDao {

    @PersistenceContext
    private EntityManager em;

    public Collection loadProductsByCategory(String category) {
       Query query = em.createQuery("from Product as p where p.category = :category");
       query.setParameter("category", category);
       return query.getResultList();
    }
}
```

Note that the `@PersistenceContext` annotation has an optional attribute `type`, which defaults to `PersistenceContextType.TRANSACTION`. This default is what you need to receive a "shared EntityManager" proxy. The alternative, `PersistenceContextType.EXTENDED`, is a completely different affair: This results in a so-called "extended EntityManager", which is *not thread-safe* and hence must not be used in a concurrently accessed component such as a Spring-managed singleton bean. Extended EntityManagers are only supposed to be used in stateful components that, for example, reside in a session, with the lifecycle of the EntityManager not tied to a current transaction but rather being completely up to the application.

Method and Field level Injection

Annotations that indicate dependency injections (such as `@PersistenceUnit` and `@PersistenceContext`) can be applied on field or methods inside a class, therefore the expression "method/field level injection". Field-level annotations concise and easier to use while method-level allow for processing the injected dependency. In both cases the member visibility (public, protected, private) does not matter.

What about class level annotations?

On JEE 5 platform, they are used for dependency declaration and not for resource injection.

The injected `EntityManager` is Spring-managed (aware of the ongoing transaction). It is important to note that even though the new implementation prefers method level injection (of an `EntityManager` instead of an `EntityManagerFactory`), no change is required in the application context XML due to annotation usage.

The main advantage of this DAO style is that it depends on Java Persistence API; no import of any Spring class is required. Moreover, as the JPA annotations are understood, the injections are applied automatically by the Spring container. This is of course appealing from a non-invasiveness perspective, and might feel more natural to JPA developers.

12.6.4. Exception Translation

However, the DAO throws the plain `PersistenceException` exception class (which is unchecked, and so does

not have to be declared or caught) but also `IllegalArgumentException` and `IllegalStateException`, which means that callers can only treat exceptions as generally fatal - unless they want to depend on JPA's own exception structure. Catching specific causes such as an optimistic locking failure is not possible without tying the caller to the implementation strategy. This tradeoff might be acceptable to applications that are strongly JPA-based and/or do not need any special exception treatment. However, Spring offers a solution allowing exception translation to be applied transparently through the `@Repository` annotation:

```
@Repository
public class ProductDaoImpl implements ProductDao {

    // class body here...

}
```

```
<beans>

  <!-- Exception translation bean post processor -->
  <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor"/>

  <bean id="myProductDao" class="product.ProductDaoImpl"/>

</beans>
```

The postprocessor will automatically look for all exception translators (implementations of the `PersistenceExceptionTranslator` interface) and advice all beans maked with the `@Repository` annotation so that the discovered translators can intercept and apply the appropriate translation on the thrown exceptions.

In summary: DAOs can be implemented based on the plain Java Persistence API and annotations, while still being able to benefit from Spring-managed transactions, dependency injection, and transparent exception conversion (if desired) to Spring's custom exception hierarchies.

12.7. Transaction Management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
       http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

  <bean id="myTxManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    <property name="entityManagerFactory" ref="myEmf"/>
  </bean>

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

  <aop:config>
    <aop:pointcut id="productServiceMethods" expression="execution(* product.ProductService.*(..))"/>
    <aop:advisor advice-ref="txAdvice" pointcut-ref="productServiceMethods"/>
  </aop:config>

  <tx:advice id="txAdvice" transaction-manager="myTxManager">
    <tx:attributes>
      <tx:method name="increasePrice*" propagation="REQUIRED"/>
      <tx:method name="someOtherBusinessMethod" propagation="REQUIRES_NEW"/>
      <tx:method name="*" propagation="SUPPORTS" read-only="true"/>
    </tx:attributes>
  </tx:advice>
```

```
</beans>
```

Spring JPA allows a configured `JpaTransactionManager` to expose a JPA transaction to JDBC access code that accesses the same JDBC `DataSource`, provided that the registered `JpaDialect` supports retrieval of the underlying JDBC `Connection`. Out of the box, Spring provides dialects for the Toplink, Hibernate and OpenJPA JPA implementations. See the next section for details on the `JpaDialect` mechanism.

12.8. `JpaDialect`

As an advanced feature `JpaTemplate`, `JpaTransactionManager` and subclasses of `AbstractEntityManagerFactoryBean` support a custom `JpaDialect`, to be passed into the "jpaDialect" bean property. In such a scenario, the DAOs won't receive an `EntityManagerFactory` reference but rather a full `JpaTemplate` instance instead (for example, passed into `JpaDaoSupport`'s "jpaTemplate" property). A `JpaDialect` implementation can enable some advanced features supported by Spring, usually in a vendor-specific manner:

- applying specific transaction semantics (such as custom isolation level or transaction timeout)
- retrieving the transactional JDBC `Connection` (for exposure to JDBC-based DAOs)
- advanced translation of `PersistenceExceptions` to Spring `DataAccessExceptions`

This is particularly valuable for special transaction semantics and for advanced translation of exception. Note that the default implementation used (`DefaultJpaDialect`) doesn't provide any special capabilities and if the above features are required, the appropriate dialect has to be specified.

See the `JpaDialect` Javadoc for more details of its operations and how they are used within Spring's JPA support.

Part III. The Web

This part of the reference documentation covers the Spring Framework's support for the presentation tier (and specifically web-based presentation tiers).

The Spring Framework's own web framework, Spring Web MVC, is covered in the first couple of chapters. A number of the remaining chapters in this part of the reference documentation are concerned with the Spring Framework's integration with other web technologies, such as Struts and JSF (to name but two).

This section concludes with coverage of Spring's MVC portlet framework.

- Chapter 13, *Web MVC framework*
- Chapter 14, *Integrating view technologies*
- Chapter 15, *Integrating with other web frameworks*
- Chapter 16, *Portlet MVC Framework*

Chapter 13. Web MVC framework

13.1. Introduction

Spring's Web MVC framework is designed around a `DispatcherServlet` that dispatches requests to handlers, with configurable handler mappings, view resolution, locale and theme resolution as well as 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 form controller. This is a major difference to Struts.

“Open for extension...”

One of the overarching design principles in Spring Web MVC (and in Spring in general) is the “*Open for extension, closed for modification*” principle.

The reason that this principle is being mentioned here is because a number of methods in the core classes in Spring Web MVC are marked `final`. This means of course that you as a developer cannot override these methods to supply your own behavior... this is *by design* and has not been done arbitrarily to annoy.

The book 'Expert Spring Web MVC and Web Flow' by Seth Ladd and others explains this principle and the reasons for adhering to it in some depth on page 117 (first edition) in the section entitled 'A Look At Design'.

If you don't have access to the aforementioned book, then the following article may be of interest the next time you find yourself going “Gah! Why can't I override this method?” (if indeed you ever do).

1. Bob Martin, The Open-Closed Principle (PDF)

Note that you cannot add advice to final methods using Spring MVC. This means it won't be possible to add advice to for example the `AbstractController.handleRequest()` method. Refer to Section 6.6.1, “Understanding AOP proxies” for more information on AOP proxies and why you cannot add advice to final methods.

Spring Web MVC allows you to use any object as a command or form object - there is no need to implement a framework-specific interface or base class. Spring's data binding is highly flexible: for example, it treats type mismatches as validation errors that can be evaluated by the application, not as system errors. All this means that you don't need to duplicate your business objects' properties as simple, untyped strings in your form objects just to be able to handle invalid submissions, or to convert the Strings properly. Instead, it is often preferable to bind directly to your business objects. This is another major difference to Struts which is built around required base classes such as `Action` and `ActionForm`.

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 only by making them bean properties of the respective `Action` class. Finally, the same `Action` instance that

handles the request is used for evaluation and form population in the view. Thus, reference data needs to be modeled as bean properties of the `Action` too. These are (arguably) too many roles for one object.

Spring's view resolution is extremely flexible. A `Controller` implementation can even write a view directly to the response (by returning `null` for the `ModelAndView`). In the normal case, a `ModelAndView` instance consists of a view name and a model `Map`, which contains bean names and corresponding objects (like a command or form, containing reference data). View name resolution is highly configurable, either via bean names, via a properties file, or via your own `ViewResolver` implementation. The fact that the model (the M in MVC) is based on the `Map` interface allows for the complete abstraction of the view technology. Any renderer can be integrated directly, whether JSP, Velocity, or any other rendering technology. The model `Map` is simply transformed into an appropriate format, such as JSP request attributes or a Velocity template model.

13.1.1. Pluggability of other MVC implementations

There are several reasons why some projects will prefer to use other MVC implementations. Many teams expect to leverage their existing investment in skills and tools. In addition, there is a large body of knowledge and experience avalailable for the Struts framework. 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 MVC 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 MVC 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, so no dedicated integration is necessary. From the web layer's point of view, you'll simply use Spring as a library, with the root application context instance as the 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 scenario, it just addresses the many areas that the pure web MVC 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, for example, the transaction abstraction with JDBC or Hibernate.

13.1.2. Features of Spring Web MVC

Spring WebFlow

Spring Web Flow (SWF) aims to be the best solution for the management of web application page flow.

SWF integrates with existing frameworks like Spring MVC, Struts, and JSF, in both servlet and portlet environments. If you have a business process (or processes) that would benefit from a conversational model as opposed to a purely request model, then SWF may be the solution.

SWF allows you to capture logical page flows as self-contained modules that are reusable in different situations, and as such is ideal for building web application modules that guide the user through controlled navigations that drive business processes.

For more information about SWF, consult the Spring WebFlow site.

Spring's web module provides a wealth of unique web support features, including:

• Clear separation of roles - controller, validator, command object, form object, model object,

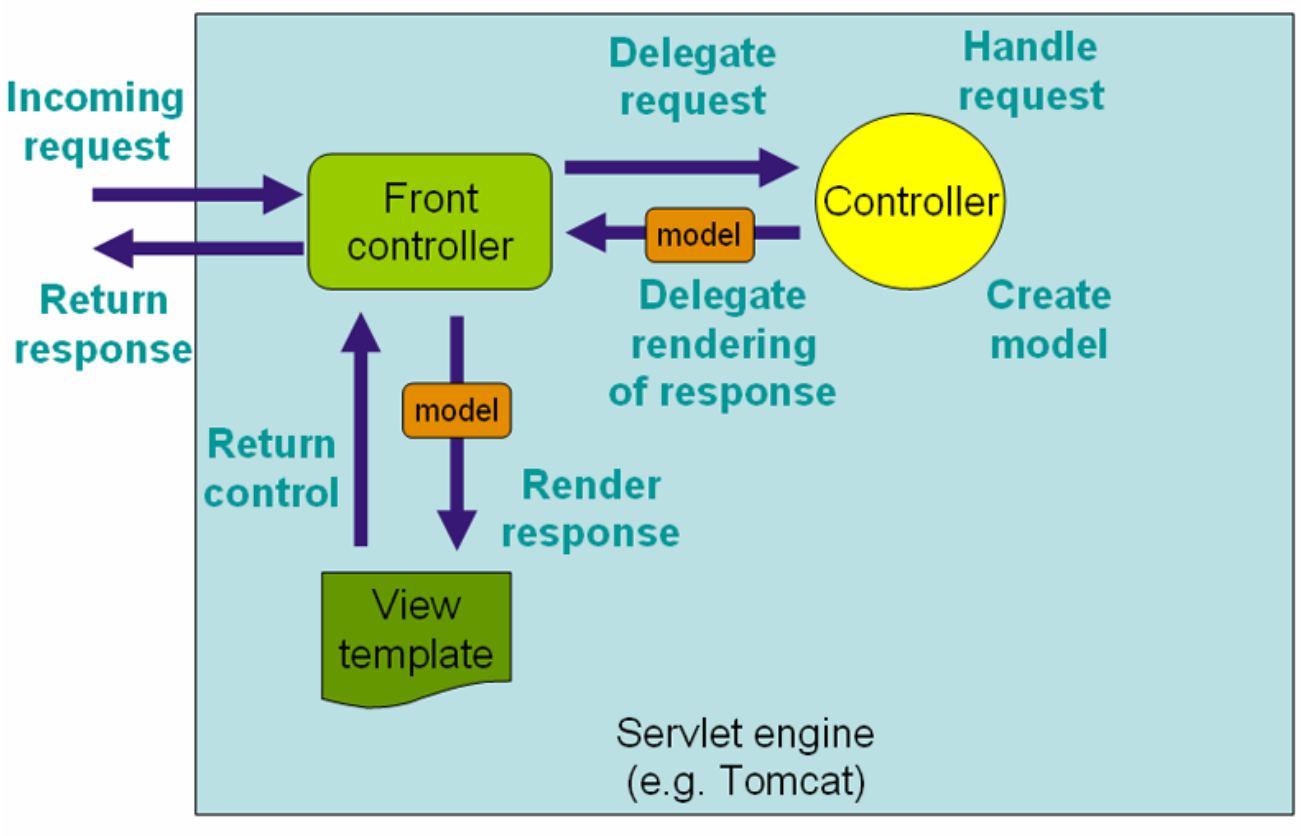
`DispatcherServlet`, handler mapping, view resolver, etc. Each role can be fulfilled by a specialized object.

- Powerful and straightforward configuration of both framework and application classes as JavaBeans, including easy referencing across contexts, such as 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 a single controller 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 order to extend a particular framework base class.
- 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 and view resolution - handler mapping and view resolution strategies range from simple URL-based configuration, to sophisticated, purpose-built resolution strategies. This is more flexible than some web MVC frameworks which mandate a particular technique.
- Flexible model transfer - model transfer via a name/value `Map` supports easy integration with any view technology.
- Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, etc.
- A simple yet powerful JSP tag library known as the Spring tag library that provides support for features such as data binding and themes. The custom tags allow for maximum flexibility in terms of markup code. For information on the tag library descriptor, see the appendix entitled Appendix D, *spring.tld*
- A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier. For information on the tag library descriptor, see the appendix entitled Appendix E, *spring-form.tld*
- Beans whose lifecycle is scoped to the current HTTP request or HTTP `Session`. This is not a specific feature of Spring MVC itself, but rather of the `WebApplicationContext` container(s) that Spring MVC uses. These bean scopes are described in detail in the section entitled Section 3.4.4, “The other scopes”

13.2. The `DispatcherServlet`

Spring's web MVC framework is, like many other web MVC frameworks, request-driven, designed around a central servlet that dispatches requests to controllers and offers other functionality facilitating the development of web applications. Spring's `DispatcherServlet` however, does more than just that. It is completely integrated with the Spring IoC container and as such allows you to use every other feature that Spring has.

The request processing workflow of the Spring Web MVC `DispatcherServlet` is illustrated in the following diagram. The pattern-savvy reader will recognize that the `DispatcherServlet` is an expression of the “Front Controller” design pattern (this is a pattern that Spring Web MVC shares with many other leading web frameworks).



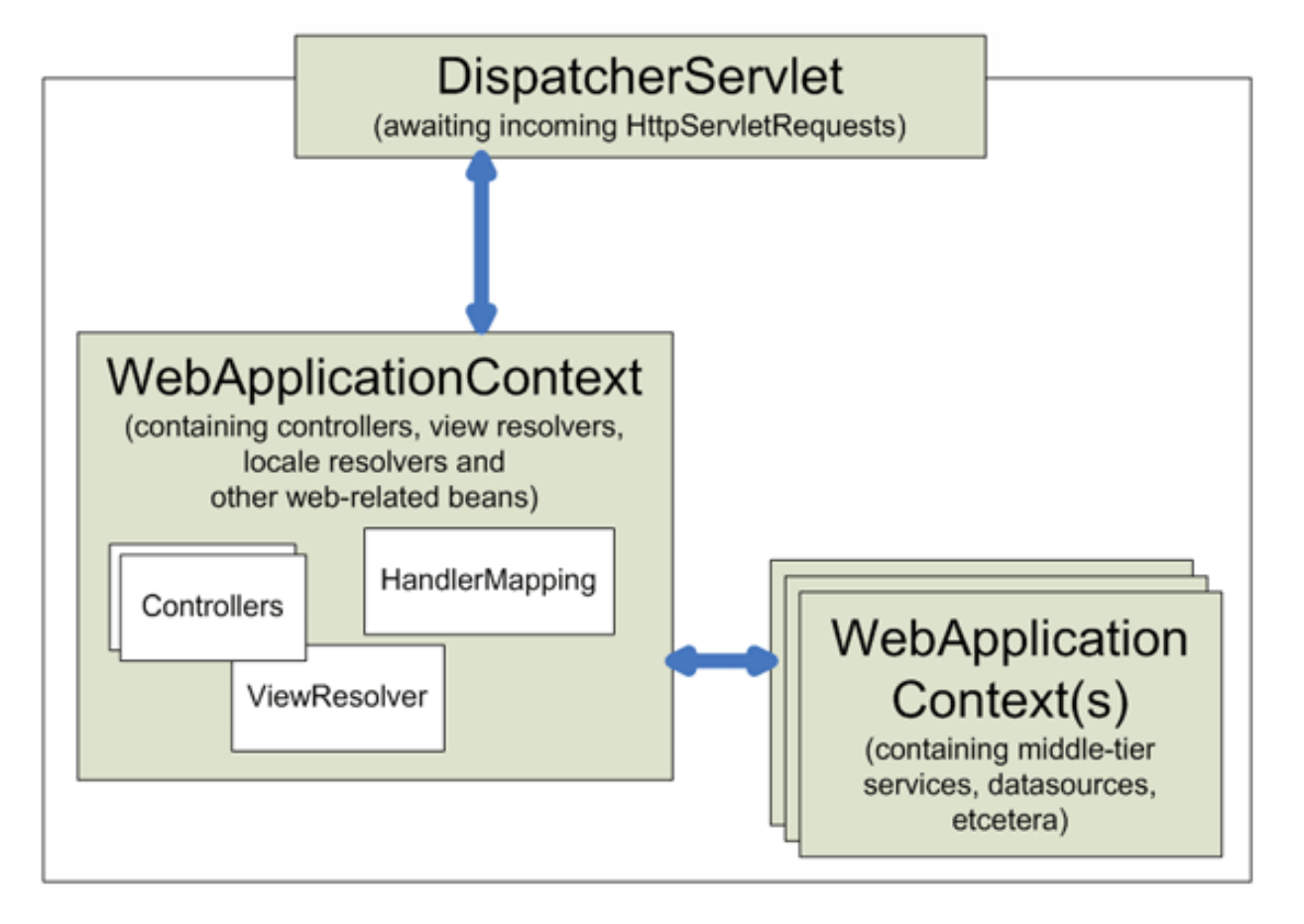
The requesting processing workflow in Spring Web MVC (high level)

The `DispatcherServlet` *is* an actual `Servlet` (it inherits from the `HttpServlet` base class), and as such is declared in the `web.xml` of your web application. Requests that you want the `DispatcherServlet` to handle will have to be mapped using a URL mapping in the same `web.xml` file. This is standard J2EE servlet configuration; an example of such a `DispatcherServlet` declaration and mapping can be found below.

```
<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 `'example' DispatcherServlet`. This is only the first step in setting up Spring Web MVC... the various beans used by the Spring Web MVC framework (over and above the `DispatcherServlet` itself) now need to be configured.

As detailed in the section entitled Section 3.8, “The `ApplicationContext`”, `ApplicationContext` instances in Spring can be scoped. In the web MVC framework, each `DispatcherServlet` has its own `WebApplicationContext`, which inherits all the beans already defined in the root `WebApplicationContext`. These inherited beans defined can be overridden in the servlet-specific scope, and new scope-specific beans can be defined local to a given servlet instance.



Context hierarchy in Spring Web MVC

The framework will, on initialization of a `DispatcherServlet`, *look for a file named* `[servlet-name]-servlet.xml` in the `WEB-INF` directory of your web application and create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

Consider the following `DispatcherServlet` servlet configuration (in the `'web.xml'` file.)

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

With the above servlet configuration in place, you will need to have a file called `'/WEB-INF/golfing-servlet.xml'` in your application; this file will contain all of your *Spring Web MVC-specific* components (beans). The exact location of this configuration file can be changed via a servlet initialization parameter (see below for details).

The `WebApplicationContext` is an extension of the plain `ApplicationContext` that has some extra features necessary for web applications. It differs from a normal `ApplicationContext` in that it is capable of resolving themes (see Section 13.7, “Using themes”), and that it knows which servlet it is associated with (by having a link to the `ServletContext`). The `WebApplicationContext` is bound in the `ServletContext`, and by using static methods on the `RequestContextUtils` class you can always lookup the `WebApplicationContext` in case

you need access to 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. These beans are included in the Spring framework and can be configured in the `WebApplicationContext`, just as any other bean would 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, sensible defaults are provided so you don't (initially) have to worry about configuring them.

Table 13.1. Special beans in the `WebApplicationContext`

Bean type	Explanation
Controllers	Controllers are the components that form the `'C'` part of the MVC.
Handler mappings	Handler mappings handle the execution of a list of pre- and post-processors and controllers that will be executed if they match certain criteria (for instance a matching URL specified with the controller)
View resolvers	View resolvers are components capable of resolving view names to views
Locale resolver	A locale resolver is a component capable of resolving the locale a client is using, in order to be able to offer internationalized views
Theme resolver	A theme resolver is capable of resolving themes your web application can use, for example, to offer personalized layouts
multipart file resolver	A multipart file resolver offers the functionality to process file uploads from HTML forms
Handler exception resolver(s)	Handler exception resolvers offer functionality to map exceptions to views or implement other more complex exception handling code

When a `DispatcherServlet` is set up for use and a request comes in for that specific `DispatcherServlet`, said `DispatcherServlet` starts processing the request. The list below describes the complete process a request goes through when handled by a `DispatcherServlet`:

1. The `WebApplicationContext` is searched for and bound in the request as an attribute in order for the controller and other elements in the process to use. 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 process resolve the locale to use when processing the request (rendering the view, preparing data, etc.) If you don't use the resolver, it won't affect anything, so if you don't need locale resolving, you don't have to use it.

3. The theme resolver is bound to the request to let elements such as views determine which theme to use. The theme resolver does not affect anything if you don't use it, so if you don't need themes you can just ignore it.

4. If a multipart resolver is specified, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a `MultipartHttpServletRequest` for further processing by other elements in the process. (See the section entitled Section 13.8.2, “Using the `MultipartResolver`” for further information about multipart handling).

5. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) will be executed in order to prepare a model (for rendering).

6. If a model is returned, the view is rendered. If no model is returned (which could be due to a pre- or postprocessor intercepting the request, for example, for security reasons), no view is rendered, since the request could already have been fulfilled.

Exceptions that are thrown during processing of the request get picked up by any of the handler exception resolvers that are declared in the `WebApplicationContext`. Using these exception resolvers allows you to define custom behaviors 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 straightforward: the `DispatcherServlet` will first lookup an appropriate handler mapping and test if the handler that is found *implements the interface* `LastModified` interface. If so, the value of the `long getLastModified(request)` method of the `LastModified` interface is returned to the client.

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

Table 13.2. `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 `[servlet-name]-servlet`.

13.3. Controllers

The notion of a controller is part of the MVC design pattern (more specifically it is the *'C'* in MVC. Controllers provide access to the application behavior which is typically defined by a service interface. Controllers interpret user input and transform such 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 form-specific controllers, command-based controllers, and controllers that execute wizard-style logic, to name but a few.

Spring's basis for the controller architecture is the `org.springframework.web.servlet.mvc.Controller` interface, the source code for 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 defines a single method that is responsible for handling a request and returning an appropriate model and view. These three concepts are the basis for the Spring MVC implementation - `ModelAndView` and `Controller`. While the `Controller` interface is quite abstract, Spring offers a lot of `Controller` implementations out of the box that already contain a lot of the functionality you might need. The `Controller` interface just defines the most basic responsibility required of every controller; namely handling a request and returning a model and a view.

13.3.1. `AbstractController` and `WebContentGenerator`

To provide a basic infrastructure, all of Spring's various `Controller` inherit from `AbstractController`, a class offering caching support and, for example, the setting of the mimetype.

Table 13.3. 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 (expedited by the throwing of a `ServletException`).
`requiresSession`	indicates whether or not this controller requires a HTTP session to do its work. If a session is not present when such a controller receives a request, the user is informed of this by a `ServletException` being thrown.
`synchronizeSession`	use this if you want handling by this controller to be synchronized on the user's HTTP session.
`cacheSeconds`	when you want a controller to generate a caching directive in the HTTP response, specify a positive integer here. By default the value of this property is set to *-1* so no caching directives will be included in the generated response.
`useExpiresHeader`	tweaks your controllers to specify the HTTP 1.0 compatible *"Expires"* header in the generated response. By default the value of this property is `true`.
`useCacheHeader`	tweaks your controllers to specify the HTTP 1.1 compatible *"Cache-Control"* header in the generated response. By default the value of this property is `true`.

When using the `AbstractController` as the baseclass for your controllers you only have to override the `handleRequestInternal(HttpServletRequest, HttpServletResponse)` method, implement your logic, and return a `ModelAndView` object. Here is short example consisting of a class and a declaration in the web application context.

```
package samples;

public class SampleController extends AbstractController {

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

        ModelAndView mav = new ModelAndView("hello");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}
```

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

The above class and the declaration in the web application context is all you need besides setting up a handler mapping (see the section entitled Section 13.4, “Handler mappings”) to get this very simple controller working. This controller will generate caching directives telling the client to cache things for 2 minutes before rechecking. This controller also returns a hard-coded view (which is typically considered bad practice).

13.3.2. Other simple controllers

Although you can extend `AbstractController`, Spring provides a number of concrete implementations which offer functionality that is commonly used in simple MVC applications. The `ParameterizableViewController` is basically the same as the example above, except for the fact that you can specify the view name that it will return in the web application context (and thus remove the need to hard-code the viewname in the Java class).

The `UrlFilenameViewController` inspects the URL and retrieves the filename of the file request and uses that as a viewname. For example, the filename of `http://www.springframework.org/index.html` request is `index`.

13.3.3. The `MultiActionController`

Spring offers a multi-action controller with which you aggregate multiple actions into one controller, thus grouping functionality together. The multi-action controller lives in a separate package - `org.springframework.web.servlet.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 have a lot of common functionality in one controller, but want to have multiple entry points to the controller, for example, to tweak behavior.

Table 13.4. Features offered by the `MultiActionController`

Feature	Explanation
`delegate`	there are two usage-scenarios for the `MultiActionController`. Either you subclass the `MultiActionController` and specify the methods that will be resolved by the `MethodNameResolver` on the subclass (in which case you don't need to set the delegate), or you define a delegate object, on which methods resolved by the `MethodNameResolver` will be invoked. If you choose this scenario, you will have to define the delegate using this configuration parameter as a collaborator.
`methodNameResolver`	the `MultiActionController` needs a strategy to resolve the method it has to invoke, based on the incoming request. This strategy is defined by the `MethodNameResolver` interface; the `MultiActionController` exposes a property sp that you can supply a resolver that is capable of doing that.

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

```
// anyMeaningfulName can be replaced by any methodname
public [ModelAndView | Map | void] anyMeaningfulName(HttpServletRequest, HttpServletResponse [, Exception | AnyC
```

Please note that method overloading is *not* allowed since it would confuse the `MultiActionController`. Furthermore, you can define *exception handlers* capable of handling exceptions that are thrown by the methods you specify.

The (optional) `Exception` argument can be *any* exception, as long as it's a subclass of `java.lang.Exception` or `java.lang.RuntimeException`. The (optional) `AnyObject` argument can be *any* class. Request parameters will be bound onto this object for convenient consumption.

Find below some examples of valid `MultiActionController` method signatures.

The standard signature (mirrors the `Controller` interface method).

```
public ModelAndView doRequest(HttpServletRequest, HttpServletResponse)
```

This signature accepts a `Login` argument that will be populated (bound) with parameters stripped from the request

```
public ModelAndView doLogin(HttpServletRequest, HttpServletResponse, Login)
```

The signature for an `Exception` handling method.

```
public ModelAndView processException(HttpServletRequest, HttpServletResponse, IllegalArgumentException)
```

This signature has a void return type (see the section entitled Section 13.11, “Convention over configuration” below).

```
public void goHome(HttpServletRequest, HttpServletResponse)
```

This signature has a `Map` return type (see the section entitled Section 13.11, “Convention over configuration” below).

```
public Map doRequest(HttpServletRequest, HttpServletResponse)
```

The `MethodNameResolver` is responsible for resolving method names based on the request coming in. Find below details about the three `MethodNameResolver` implementations that Spring provides out of the box.

- `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). The `paramName` property specifies the request parameter that is to be inspected).
- `InternalPathMethodNameResolver` - retrieves the filename from the request 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 method names. 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`, so if the properties contained `/**/welcom?.html`, it would also have worked!

Here are a couple of examples. First, an example 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"/>
</bean>

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

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

## together with

public class SampleDelegate {

    public ModelAndView retrieveIndex(HttpServletRequest req, HttpServletResponse resp) {

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

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

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

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

13.3.4. Command controllers

Spring's *command controllers* are a fundamental part of the Spring Web MVC package. Command controllers provide a way to interact with data objects and dynamically bind parameters from the `HttpServletRequest` to the data object specified. They perform a somewhat similar role to the Struts `ActionForm`, but in Spring, your data objects don't have to implement a framework-specific interface. First, lets examine what command controllers are available straight out of the box.

- `AbstractCommandController` - a command controller you can use to create your own command controller, capable of binding request parameters to a data object you specify. This class does not offer form functionality; it does however offer validation features and lets you specify in the controller itself what to do with the command object that has been populated with request parameter values.
- `AbstractFormController` - an abstract controller offering form submission support. Using this controller you can model forms and populate them using a command object you retrieve in the controller. After a user has filled the form, the `AbstractFormController` binds the fields, validates the command object, and hands the object back to the controller to take the appropriate action. Supported features are: invalid form submission (resubmission), validation, and normal form workflow. You implement methods to determine which views are used for form presentation and success. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the application context.
- `SimpleFormController` - a form controller that provides even more support when creating a form with a corresponding command object. 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 form submission has succeeded, and more.

- `AbstractWizardFormController` - as the class name suggests, this is an abstract class - your wizard controller should extend it. This means you have to implement the `validatePage()`, `processFinish()` and `processCancel()` methods.

  You probably also want to write a contractor, which should at the very least call `setPages()` and `setCommandName()`. The former takes as its argument an array of type String. This array is the list of views which comprise your wizard. The latter takes as its argument a String, which will be used to refer to your command object from within your views.

  As with any instance of `AbstractFormController`, you are required to use a command object - a JavaBean which will be populated with the data from your forms. You can do this in one of two ways: either call `setCommandClass()` from the constructor with the class of your command object, or implement the `formBackingObject()` method.

  `AbstractWizardFormController` has a number of concrete methods that you may wish to override. Of these, the ones you are likely to find most useful are: `referenceData(..)` which you can use to pass model data to your view in the form of a `Map`; `getTargetPage()` if your wizard needs to change page order or omit pages dynamically; and `onBindAndValidate()` if you want to override the built-in binding and validation workflow.

  Finally, it is worth pointing out the `setAllowDirtyBack()` and `setAllowDirtyForward()`, which you can call from `getTargetPage()` to allow users to move backwards and forwards in the wizard even if validation fails for the current page.

  For a full list of methods, see the Javadoc for `AbstractWizardFormController`. There is an implemented example of this wizard in the jPetStore included in the Spring distribution: `org.springframework.samples.jpetstore.web.spring.OrderFormController`.

13.4. Handler mappings

Using a handler mapping you can map incoming web requests to appropriate handlers. There are some handler mappings you can use out of the box, 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`, which must contain the handler that matches the incoming request, and may also contain a list of handler interceptors that are 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. Then 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 into custom `HandlerMapping`s. 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.

This section describes two of Spring's most commonly used handler mappings. They both extend the `AbstractHandlerMapping` and share the following properties:

- `interceptors`: the list of interceptors to use. `HandlerInterceptors` are discussed in Section 13.4.3, “Intercepting requests - the `HandlerInterceptor` interface”.
- `defaultHandler`: the default handler to use, when this handler mapping does not result in a matching handler.
- `order`: based on the value of the order property (see the `org.springframework.core.Ordered` interface),

Spring will sort all handler mappings available in the context and apply the first matching handler.

- `alwaysUseFullPath`: if this property is set to `true`, Spring will use the full path within the current servlet context to find an appropriate handler. If this property is set to `false` (the default), the path within the current servlet mapping will be used. For example, if a servlet is mapped using `/testing/*` and the `alwaysUseFullPath` property is set to true, `/testing/viewPage.html` would be used, whereas if the property is set to false, `/viewPage.html` would be used.
- `urlPathHelper`: using this property, you can tweak the UrlPathHelper used when inspecting URLs. Normally, you shouldn't have to change the default value.
- `urlDecode`: the default value for this property is `false`. The `HttpServletRequest` returns request URLs and URIs that are *not* decoded. If you do want them to be decoded before a `HandlerMapping` uses them to find an appropriate handler, you have to set this to `true` (note that this requires JDK 1.4). The decoding method uses either the encoding specified by the request or the default ISO-8859-1 encoding scheme.
- `lazyInitHandlers`: allows for lazy initialization of *singleton* handlers (prototype handlers are always lazily initialized). Default value is `false`.

(*Note: the last four properties are only available to subclasses of* `org.springframework.web.servlet.handler.AbstractUrlHandlerMapping`).

13.4.1. `BeanNameUrlHandlerMapping`

A very simple, but very powerful handler mapping is the `BeanNameUrlHandlerMapping`, which maps incoming HTTP requests to names of beans, defined in the web application context. Let's say we want to enable a user to insert an account and we've already provided an appropriate form controller (see Section 13.3.4, “Command controllers” for more information on command- and form controllers) and a JSP view (or Velocity template) that renders the form. When using the `BeanNameUrlHandlerMapping`, we could map the HTTP request with the URL `http://samples.com/editaccount.form` to the appropriate form `Controller` 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"/>
    <property name="successView" value="account-created"/>
    <property name="commandName" value="account"/>
    <property name="commandClass" value="samples.Account"/>
  </bean>
<beans>
```

All incoming requests for the URL `/editaccount.form` will now be handled by the form `Controller` 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 web application context (as indicated above). By default, if no handler mapping can be found in the context, the `DispatcherServlet` creates a `BeanNameUrlHandlerMapping` for you!

13.4.2. `SimpleUrlHandlerMapping`

A further - and much more powerful handler mapping - is the `SimpleUrlHandlerMapping`. This mapping is configurable in the application context and has Ant-style path matching capabilities (see the Javadoc for the `org.springframework.util.PathMatcher` class). Here is an example:

```
<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>

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

The above `web.xml` configuration snippet enables all requests ending with .html and `.form` to be handled by the sample dispatcher servlet.

```
<beans>

    <!-- no 'id' required, HandlerMapping beans are automatically detected by the DispatcherServlet -->
    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                /*/account.form=editAccountFormController
                /*/editaccount.form=editAccountFormController
                /ex/view*.html=helpController
                /**/help.html=helpController
            </value>
        </property>
    </bean>

    <bean id="helpController"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

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

This handler mapping routes requests for `'help.html'` in any directory to the `'helpController'`, which is a `UrlFilenameViewController` (more about controllers can be found in the section entitled Section 13.3,

“Controllers”). Requests for a resource beginning with `'view'`, and ending with `'.html'` in the directory `'ex'` will be routed to the `'helpController'`. Two further mappings are also defined for `'editAccountFormController'`.

13.4.3. Intercepting requests - the `HandlerInterceptor` interface

Spring's handler mapping mechanism has the notion of handler interceptors, that can be extremely useful when you want to apply specific functionality to certain requests, for example, checking for a principal.

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*. These three methods should provide enough flexibility to do all kinds of pre- and post-processing.

The `preHandle(..)` method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns `true`, the handler execution chain will continue, when it returns false, the `DispatcherServlet` assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue 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">
            <value>
                /*.form=editAccountFormController
                /*.view=editAccountFormController
            </value>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
          class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime" value="9"/>
        <property name="closingTime" value="18"/>
    </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 redirected to a static html file, saying, for example, he can only access the website during office hours.

As you can see, Spring has an adapter class (the cunningly named `HandlerInterceptorAdapter`) to make it easier to extend the `HandlerInterceptor` interface.

13.5. Views and resolving them

All MVC frameworks for web applications provide a way to address views. Spring provides view resolvers, which enable you to render models in a browser without tying you to a specific view technology. Out of the box, Spring enables you to use JSPs, Velocity templates and XSLT views, for example. The section entitled Chapter 14, *Integrating view technologies* has details of how to integrate and use a number of disparate view technologies.

The two interfaces which are important to the way Spring handles views are `ViewResolver` and `View`. The `ViewResolver` provides a mapping between view names and actual views. The `View` interface addresses the preparation of the request and hands the request over to one of the view technologies.

13.5.1. Resolving views - the `ViewResolver` interface

As discussed in the section entitled Section 13.3, “Controllers”, all controllers in the Spring Web MVC framework return a `ModelAndView` instance. Views in Spring are addressed by a view name and are resolved by a view resolver. Spring comes with quite a few view resolvers. We'll list most of them and then provide a couple of examples.

Table 13.5. View resolvers

`ViewResolver`	Description
`AbstractCachingViewResolver`	An abstract view resolver which takes care of caching views. Often views need preparation before they can be used, extending this view resolver provides caching of views.
`XmlViewResolver`	An implementation of `ViewResolver` that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is `/WEB-INF/views.xml`.
`ResourceBundleViewResolver`	An 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. The default file name is `views.properties`.

ViewResolver	Description
UrlBasedViewResolver	A simple implementation of the ViewResolver interface that effects the direct resolution of symbolic view names to URLs, without an 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	A convenience subclass of UrlBasedViewResolver that supports InternalResourceView (i.e. Servlets and JSPs), and subclasses such as JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass(..). See the Javadocs for the UrlBasedViewResolver class for details.
VelocityViewResolver / FreeMarkerViewResolver	A convenience subclass of UrlBasedViewResolver that supports VelocityView (i.e. Velocity templates) or FreeMarkerView respectively and custom subclasses of them.

As an example, when using JSP for a view technology you can use the UrlBasedViewResolver. This view resolver translates a view name 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/"/>
    <property name="suffix" value=".jsp"/>
</bean>
```

When returning test as a viewname, this view resolver will hand the request over to the RequestDispatcher that will send the request to /WEB-INF/jsp/test.jsp.

When mixing different view technologies in a web application, you can use the ResourceBundleViewResolver:

```
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
    <property name="defaultParentView" value="parentView"/>
</bean>
```

The ResourceBundleViewResolver inspects the ResourceBundle identified by the basename, and for each view it is supposed to resolve, it uses the value of the property [viewname].class as the view class and the value of the property [viewname].url as the view url. As you can see, you can identify a parent view, from which all views in the properties file sort of extend. This way you can specify a default view class, for example.

A note on caching - subclasses of AbstractCachingViewResolver cache view instances they have resolved. This greatly improves performance when using certain view technologies. It's possible to turn off the cache, by setting the cache property to false. Furthermore, if you have the requirement to be able to refresh a certain view at runtime (for example when a Velocity template has been modified), you can use the removeFromCache(String viewName, Locale loc) method.

13.5.2. Chaining ViewResolvers

Spring supports more than just one view resolver. This allows you to chain resolvers and, for example, override specific views in certain circumstances. Chaining view resolvers is pretty straightforward - just add more than one resolver to your application context and, if necessary, set the order property to specify an order. Remember, the higher the order property, the later the view resolver will be positioned in the chain.

In the following example, the chain of view resolvers consists of two resolvers, a `InternalResourceViewResolver` (which is always automatically positioned as the last resolver in the chain) and an `XmlViewResolver` for specifying Excel views (which are not supported by the `InternalResourceViewResolver`):

```
<bean id="jspViewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.JstlView"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>

<bean id="excelViewResolver" class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

<!-- in views.xml -->

<beans>
  <bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

If a specific view resolver does not result in a view, Spring will inspect the context to see if other view resolvers are configured. If there are additional view resolvers, it will continue to inspect them. If not, it will throw an `Exception`.

You have to keep something else in mind - the contract of a view resolver mentions that a view resolver *can* return null to indicate the view could not be found. Not all view resolvers do this however! This is because in some cases, the resolver simply cannot detect whether or not the view exists. For example, the `InternalResourceViewResolver` uses the `RequestDispatcher` internally, and dispatching is the only way to figure out if a JSP exists - this can only be done once. The same holds for the `VelocityViewResolver` and some others. Check the Javadoc for the view resolver to see if you're dealing with a view resolver that does not report non-existing views. As a result of this, putting an `InternalResourceViewResolver` in the chain in a place other than the last, will result in the chain not being fully inspected, since the `InternalResourceViewResolver` will *always* return a view!

13.5.3. Redirecting to views

As has been mentioned, a controller normally returns a logical view name, which a view resolver resolves to a particular view technology. For view technologies such as JSPs that are actually processed via the Servlet/JSP engine, this is normally handled via `InternalResourceViewResolver` / `InternalResourceView` which will ultimately end up issuing an internal forward or include, via the Servlet API's `RequestDispatcher.forward(..)` or `RequestDispatcher.include()`. For other view technologies, such as Velocity, XSLT, etc., the view itself produces the content on the response stream.

It is sometimes desirable to issue an HTTP redirect back to the client, before the view is rendered. This is desirable for example when one controller has been called with `POST`ed data, and the response is actually a delegation to another controller (for example on a successful form submission). In this case, a normal internal forward will mean the other controller will also see the same `POST` data, which is potentially problematic if it can confuse it with other expected data. Another reason to do a redirect before displaying the result is that this will eliminate the possibility of the user doing a double submission of form data. The browser will have sent the initial `POST`, will have seen a redirect back and done a subsequent `GET` because of that, and thus as far as it is concerned, the current page does not reflect the result of a `POST`, but rather of a `GET`, so there is no way the user can accidentally re-`POST` the same data by doing a refresh. The refresh would just force a `GET` of the result page, not a resend of the initial `POST` data.

13.5.3.1. `RedirectView`

One way to force a redirect as the result of a controller response is for the controller to create and return an instance of Spring's `RedirectView`. In this case, `DispatcherServlet` will not use the normal view resolution mechanism, but rather as it has been given the (redirect) view already, will just ask it to do its work.

The `RedirectView` simply ends up issuing an `HttpServletResponse.sendRedirect()` call, which will come back to the client browser as an HTTP redirect. All model attributes are simply exposed as HTTP query parameters. This does mean that the model must contain only objects (generally Strings or convertible to Strings) which can be readily converted to a string-form HTTP query parameter.

If using `RedirectView` and the view is created by the controller itself, it is preferable for the redirect URL to be injected into the controller so that it is not baked into the controller but configured in the context along with the view names.

13.5.3.2. The `redirect:` prefix

While the use of `RedirectView` works fine, if the controller itself is creating the `RedirectView`, there is no getting around the fact that the controller is aware that a redirection is happening. This is really suboptimal and couples things too tightly. The controller should not really care about how the response gets handled... it should generally think only in terms of view names that have been injected into it.

The special `redirect:` prefix allows this to be achieved. If a view name is returned which has the prefix redirect:, then `UrlBasedViewResolver` (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The net effect is the same as if the controller had returned a `RedirectView`, but now the controller itself can deal just in terms of logical view names. A logical view name such as `redirect:/my/response/controller.html` will redirect relative to the current servlet context, while a name such as `redirect:http://myhost.com/some/arbitrary/path.html` will redirect to an absolute URL. The important thing is that as long is this redirect view name is injected into the controller like any other logical view name, the controller is not even aware that redirection is happening.

13.5.3.3. The `forward:` prefix

It is also possible to use a special `forward:` prefix for view names that will ultimately be resolved by `UrlBasedViewResolver` and subclasses. All this does is create an `InternalResourceView` (which ultimately does a `RequestDispatcher.forward()`) around the rest of the view name, which is considered a URL. Therefore, there is never any use in using this prefix when using `InternalResourceViewResolver` / `InternalResourceView` anyway (for JSPs for example), but it's of potential use when you are primarily using another view technology, but still want to force a forward to happen to a resource to be handled by the Servlet/JSP engine. (Note that you may also chain multiple view resolvers, instead.)

As with the `redirect:` prefix, if the view name with the prefix is just injected into the controller, the controller does not have to be aware that anything special is happening in terms of handling the response.

13.6. Using locales

Most parts of Spring's architecture support internationalization, just as the Spring web MVC framework does. `DispatcherServlet` enables you to automatically resolve messages using the client's locale. This is done with `LocaleResolver` objects.

When a request comes in, the `DispatcherServlet` looks for a locale resolver and if it finds one it tries to use it to set the locale. Using the `RequestContext.getLocale()` method, you can always retrieve the locale that was resolved by the locale resolver.

Besides the automatic locale resolution, you can also attach an interceptor to the handler mapping (see Section 13.4.3, “Intercepting requests - the `HandlerInterceptor` interface” for more information on handler mapping interceptors), to change the locale under specific circumstances, based on a parameter in the request, for example.

Locale resolvers and interceptors are all defined in the `org.springframework.web.servlet.i18n` package, and are configured in your application context in the normal way. Here is a selection of the locale resolvers included in Spring.

13.6.1. `AcceptHeaderLocaleResolver`

This locale resolver inspects 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.

13.6.2. `CookieLocaleResolver`

This locale resolver inspects a `Cookie` that might exist on the client, to see if a locale is 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. Find below an example of defining a `CookieLocaleResolver`.

```
<bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">

    <property name="cookieName" value="clientlanguage"/>

    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
    <property name="cookieMaxAge" value="100000">

</bean>
```

Table 13.6. `CookieLocaleResolver` properties

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 -1 is specified, the cookie will not be persisted. It will only be available 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 it.

13.6.3. `SessionLocaleResolver`

The `SessionLocaleResolver` allows you to retrieve locales from the session that might be associated with the user's request.

13.6.4. `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 13.4, “Handler mappings”). It will detect a parameter in the request and change the locale (it calls `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"/>
</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 bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <value>/**/*.view=someController</value>
    </property>
</bean>
```

All calls to all `*.view` resources containing a parameter named `siteLanguage` will now change the locale. So a request for the following URL, `http://www.sf.net/home.view?siteLanguage=nl` will change the site language to Dutch.

13.7. Using themes

13.7.1. Introduction

The *theme* support provided by the Spring web MVC framework enables you to further enhance the user experience by allowing the look and feel of your application to be *themed*. A theme is basically a collection of static resources affecting the visual style of the application, typically style sheets and images.

13.7.2. Defining themes

When you want to use themes in your web application you'll have to set up a `org.springframework.ui.context.ThemeSource`. The `WebApplicationContext` interface extends `ThemeSource` but delegates its responsibilities to a dedicated implementation. By default the delegate will be a `org.springframework.ui.context.support.ResourceBundleThemeSource` that loads properties files from the root of the classpath. If you want to use a custom `ThemeSource` implementation or if you need to configure the basename prefix of the `ResourceBundleThemeSource`, you can register a bean in the application context with the reserved name "themeSource". The web application context will automatically detect that bean and start using it.

When using the `ResourceBundleThemeSource`, a theme is defined in a simple properties file. The properties file lists the resources that make up the theme. Here is an example:

```
styleSheet=/themes/cool/style.css
background=/themes/cool/img/coolBg.jpg
```

The keys of the properties are the names used to refer to the themed elements from view code. For a JSP this

would typically be done using the `spring:theme` custom tag, which is very similar to the `spring:message` tag. The following JSP fragment uses the theme defined above to customize the look and feel:

```
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<html>
   <head>
      <link rel="stylesheet" href="<spring:theme code="styleSheet"/>" type="text/css"/>
   </head>
   <body background="<spring:theme code="background"/>">
      ...
   </body>
</html>
```

By default, the `ResourceBundleThemeSource` uses an empty basename prefix. As a result the properties files will be loaded from the root of the classpath, so we'll have to put our `cool.properties` theme definition in a directory at the root of the classpath, e.g. in `/WEB-INF/classes`. Note that the `ResourceBundleThemeSource` uses the standard Java resource bundle loading mechanism, allowing for full internationalization of themes. For instance, we could have a `/WEB-INF/classes/cool_nl.properties` that references a special background image, e.g. with Dutch text on it.

13.7.3. Theme resolvers

Now that we have our themes defined, the only thing left to do is decide which theme to use. The `DispatcherServlet` will look for a bean named "themeResolver" to find out which `ThemeResolver` implementation to use. A theme resolver works in much the same way as a `LocaleResolver`. It can detect the theme that should be used for a particular request and can also alter the request's theme. The following theme resolvers are provided by Spring:

Table 13.7. `ThemeResolver` implementations

Class	Description
`FixedThemeResolver`	Selects a fixed theme, set using the "defaultThemeName" property.
`SessionThemeResolver`	The theme is maintained in the users HTTP session. It only needs to be set once for each session, but is not persisted between sessions.
`CookieThemeResolver`	The selected theme is stored in a cookie on the user-agent's machine.

Spring also provides a `ThemeChangeInterceptor`, which allows changing the theme on every request by including a simple request parameter.

13.8. Spring's multipart (fileupload) support

13.8.1. Introduction

Spring has built-in multipart support to handle fileuploads in web applications. The design for the multipart support is done with pluggable `MultipartResolver` objects, defined in the `org.springframework.web.multipart` package. Out of the box, Spring provides `MultipartResolvers` for use with *Commons FileUpload* (http://jakarta.apache.org/commons/fileupload) and *COS FileUpload* (http://www.servlets.com/cos). How uploading files is supported will be described in the rest of this chapter.

By default, no multipart handling will be done by Spring, as some developers will want to handle multiparts themselves. You will have to enable it yourself by adding a multipart resolver to the web application's context. After you have done that, each request will be inspected to see if it contains a multipart. If no multipart is found, the request will continue as expected. However, if a multipart is found in the request, the `MultipartResolver` that has been declared in your context will be used. After that, the multipart attribute in your request will be treated like any other attribute.

13.8.2. Using the `MultipartResolver`

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="maxUploadSize" value="100000"/>
</bean>
```

This is an example using 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="maxUploadSize" value="100000"/>
</bean>
```

Of course you also need to put the appropriate jars in your classpath for the multipart resolver to work. In the case of the `CommonsMultipartResolver`, you need to use `commons-fileupload.jar`; in the case of the `CosMultipartResolver`, use `cos.jar`.

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

13.8.3. Handling a file upload in a form

After the `MultipartResolver` has finished doing its job, the request will be processed like any other. To use it, you create a form with an upload field (see immediately below), then let Spring bind the file onto your form (backing object). To actually let the user upload a file, we have to create a (HTML) 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 (`enctype="multipart/form-data"`) which is necessary to let the browser know how to encode the multipart fields (do not forget this!).

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 objects you have to register a custom editor with the `ServletRequestDatabinder`. There are a couple of editors available for handling files and setting the results on an object. There's a `StringMultipartEditor` capable of converting files to Strings (using a user-defined character set) and there is a `ByteArrayMultipartEditor` which converts files to byte arrays. They function just as the `CustomDateEditor` does.

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

```
<beans>
        <!-- lets use the Commons-based implementation of the MultipartResolver interface -->
    <bean id="multipartResolver"
        class="org.springframework.web.multipart.commons.CommonsMultipartResolver"/>

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

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

</beans>
```

After that, create the controller and the actual class to hold the file property.

```
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
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

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 registers a custom editor to let Spring know how to actually convert the multipart objects the resolver has found to properties specified by the bean. In this example, nothing is done with the `byte[]` property of the bean itself, but in practice you can do whatever you want (save it in a database, mail it to somebody, etc).

An equivalent example in which a file is bound straight to a String-typed property on a (form backing) object might look like:

```
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
        String 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 a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class, new StringMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert them
    }

}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

Of course, this last example only makes (logical) sense in the context of uploading a plain text file (it wouldn't work so well in the case of uploading an image file).

The third (and final) option is where one binds directly to a `MultipartFile` property declared on the (form backing) object's class. In this case one does not need to register any custom `PropertyEditor` because there is no type conversion to be performed.

```
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
        MultipartFile 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);
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
}
```

13.9. Using Spring's form tag library

As of version 2.0, Spring provides a comprehensive set of data binding-aware tags for handling form elements when using JSP and Spring Web MVC. Each tag provides support for the set of attributes of its corresponding HTML tag counterpart, making the tags familiar and intuitive to use. The tag-generated HTML is HTML 4.01/XHTML 1.0 compliant.

Unlike other form/input tag libraries, Spring's form tag library is integrated with Spring Web MVC, giving the tags access to the command object and reference data your controller deals with. As you will see in the following examples, the form tags make JSPs easier to develop, read and maintain.

Let's go through the form tags and look at an example of how each tag is used. We have included generated HTML snippets where certain tags require further commentary.

13.9.1. Configuration

The form tag library comes bundled in `spring.jar`. The library descriptor is called `spring-form.tld`.

To use the tags from this library, add the following directive to the top of your JSP page:

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

... where `form` is the tag name prefix you want to use for the tags from this library.

13.9.2. The `form` tag

This tag renders an HTML 'form' tag and exposes a binding path to inner tags for binding. It puts the command object in the `PageContext` so that the command object can be accessed by inner tags. *All the other tags in this*

library are nested tags of the `form` tag.

Let's assume we have a domain object called `User`. It is a JavaBean with properties such as `firstName` and `lastName`. We will use it as the form backing object of our form controller which returns `form.jsp`. Below is an example of what `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

The `firstName` and `lastName` values are retrieved from the command object placed in the `PageContext` by the page controller. Keep reading to see more complex examples of how inner tags are used with the `form` tag.

The generated HTML looks like a standard form:

```
<form method="POST">
    <table>
      <tr>
          <td>First Name:</td>
          <td><input name="firstName" type="text" value="Harry"/></td>
          <td></td>
      </tr>
      <tr>
          <td>Last Name:</td>
          <td><input name="lastName" type="text" value="Potter"/></td>
          <td></td>
      </tr>
      <tr>
          <td colspan="3">
            <input type="submit" value="Save Changes" />
          </td>
      </tr>
    </table>
</form>
```

The preceding JSP assumes that the variable name of the form backing object is `'command'`. If you have put the form backing object into the model under another name (definitely a best practice), then you can bind the form to the named variable like so:

```
<form:form commandName="user">
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

13.9.3. The `input` tag

This tag renders an HTML 'input' tag with type 'text' using the bound value. For an example of this tag, see Section 13.9.2, “The `form` tag”.

13.9.4. The `checkbox` tag

This tag renders an HTML 'input' tag with type 'checkbox'.

Let's assume our `User` has preferences such as newsletter subscription and a list of hobbies. Below is an example of the `Preferences` class:

```
public class Preferences {

    private boolean receiveNewsletter;

    private String[] interests;

    private String favouriteWord;

    public boolean isReceiveNewsletter() {
        return receiveNewsletter;
    }

    public void setReceiveNewsletter(boolean receiveNewsletter) {
        this.receiveNewsletter = receiveNewsletter;
    }

    public String[] getInterests() {
        return interests;
    }

    public void setInterests(String[] interests) {
        this.interests = interests;
    }

    public String getFavouriteWord() {
        return favouriteWord;
    }

    public void setFavouriteWord(String favouriteWord) {
        this.favouriteWord = favouriteWord;
    }
}
```

The `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>Subscribe to newsletter?:</td>
            <%-- Approach 1: Property is of type java.lang.Boolean --%>
            <td><form:checkbox path="preferences.receiveNewsletter"/></td>
            <td> </td>
        </tr>

        <tr>
            <td>Interests:</td>
            <td>
                <%-- Approach 2: Property is of an array or of type java.util.Collection --%>
                Quidditch: <form:checkbox path="preferences.interests" value="Quidditch"/>
                Herbology: <form:checkbox path="preferences.interests" value="Herbology"/>
                Defence Against the Dark Arts: <form:checkbox path="preferences.interests"
                    value="Defence Against the Dark Arts"/>
            </td>
            <td> </td>
        </tr>
        <tr>
            <td>Favourite Word:</td>
            <td>
```

```
                <%-- Approach 3: Property is of type java.lang.Object --%>
                Magic: <form:checkbox path="preferences.favouriteWord" value="Magic"/>
            </td>
            <td> </td>
        </tr>
    </table>
</form:form>
```

There are 3 approaches to the `checkbox` tag which should meet all your checkbox needs.

- Approach One - When the bound value is of type `java.lang.Boolean`, the `input(checkbox)` is marked as 'checked' if the bound value is `true`. The `value` attribute corresponds to the resolved value of the `setValue(Object)` value property.
- Approach Two - When the bound value is of type `array` or `java.util.Collection`, the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` value is present in the bound `Collection`.
- Approach Three - For any other bound value type, the `input(checkbox)` is marked as 'checked' if the configured `setValue(Object)` is equal to the bound value.

Note that regardless of the approach, the same HTML structure is generated. Below is an HTML snippet of some checkboxes:

```
<tr>
    <td>Interests:</td>
    <td>
        Quidditch: <input name="preferences.interests" type="checkbox" value="Quidditch"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Herbology: <input name="preferences.interests" type="checkbox" value="Herbology"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
        Defence Against the Dark Arts: <input name="preferences.interests" type="checkbox"
            value="Defence Against the Dark Arts"/>
        <input type="hidden" value="1" name="_preferences.interests"/>
    </td>
    <td> </td>
</tr>
```

What you might not expect to see is the additional hidden field after each checkbox. When a checkbox in an HTML page is *not* checked, its value will not be sent to the server as part of the HTTP request parameters once the form is submitted, so we need a workaround for this quirk in HTML in order for Spring form data binding to work. The `checkbox` tag follows the existing Spring convention of including a hidden parameter prefixed by an underscore ("_") for each checkbox. By doing this, you are effectively telling Spring that "*the checkbox was visible in the form and I want my object to which the form data will be bound to reflect the state of the checkbox no matter what*".

13.9.5. The `radiobutton` tag

This tag renders an HTML 'input' tag with type 'radio'.

A typical usage pattern will involve multiple tag instances bound to the same property but with different values.

```
<tr>
    <td>Sex:</td>
    <td>Male: <form:radiobutton path="sex" value="M"/> <br/>
        Female: <form:radiobutton path="sex" value="F"/> </td>
    <td> </td>
</tr>
```

13.9.6. The `password` tag

This tag renders an HTML 'input' tag with type 'password' using the bound value.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" />
    </td>
</tr>
```

Please note that by default, the password value is *not* shown. If you do want the password value to be shown, then set the value of the `'showPassword'` attribute to true, like so.

```
<tr>
    <td>Password:</td>
    <td>
        <form:password path="password" value="^76525bvHGq" showPassword="true" />
    </td>
</tr>
```

13.9.7. The `select` tag

This tag renders an HTML 'select' element. It supports data binding to the selected option as well as the use of nested `option` and `options` tags.

Let's assume a `User` has a list of skills.

```
<tr>
    <td>Skills:</td>
    <td><form:select path="skills" items="${skills}"/></td>
    <td></td>
</tr>
```

If the `User's` skill were in Herbology, the HTML source of the 'Skills' row would look like:

```
<tr>
    <td>Skills:</td>
    <td><select name="skills" multiple="true">
        <option value="Potions">Potions</option>
        <option value="Herbology" selected="true">Herbology</option>
        <option value="Quidditch">Quidditch</option></select></td>
    <td></td>
</tr>
```

13.9.8. The `option` tag

This tag renders an HTML 'option'. It sets 'selected' as appropriate based on the bound value.

```
<tr>
    <td>House:</td>
    <td>
        <form:select path="house">
            <form:option value="Gryffindor"/>
            <form:option value="Hufflepuff"/>
            <form:option value="Ravenclaw"/>
            <form:option value="Slytherin"/>
        </form:select>
    </td>
</tr>
```

If the `User's` house was in Gryffindor, the HTML source of the 'House' row would look like:

```
<tr>
    <td>House:</td>
    <td>
        <select name="house">
            <option value="Gryffindor" selected="true">Gryffindor</option>
            <option value="Hufflepuff">Hufflepuff</option>
            <option value="Ravenclaw">Ravenclaw</option>
            <option value="Slytherin">Slytherin</option>
        </select>
    </td>
 </tr>
```

13.9.9. The `options` tag

This tag renders a list of HTML 'option' tags. It sets the 'selected' attribute as appropriate based on the bound value.

```
<tr>
    <td>Country:</td>
    <td>
        <form:select path="country">
            <form:option value="-" label="--Please Select"/>
            <form:options items="${countryList}" itemValue="code" itemLabel="name"/>
        </form:select>
    </td>
    <td></td>
</tr>
```

If the `User` lived in the UK, the HTML source of the 'Country' row would look like:

```
 <tr>
    <td>Country:</td>
    <tr>
        <td>Country:</td>
        <td>
            <select name="country">
                <option value="-">--Please Select</option>
                <option value="AT">Austria</option>
                <option value="UK" selected="true">United Kingdom</option>
                <option value="US">United States</option>
            </select>
        </td>
        <td></td>
    </tr>
    <td></td>
</tr>
```

As the example shows, the combined usage of an `option` tag with the `options` tag generates the same standard HTML, but allows you to explicitly specify a value in the JSP that is for display only (where it belongs) such as the default string in the example: "-- Please Select".

13.9.10. The `textarea` tag

This tag renders an HTML 'textarea'.

```
<tr>
    <td>Notes:</td>
    <td><form:textarea path="notes" rows="3" cols="20" /></td>
    <td><form:errors path="notes" /></td>
</tr>
```

13.9.11. The `hidden` tag

This tag renders an HTML 'input' tag with type 'hidden' using the bound value. To submit an unbound hidden value, use the HTML `input` tag with type 'hidden'.

```
<form:hidden path="house" />
```

If we choose to submit the 'house' value as a hidden one, the HTML would look like:

```
<input name="house" type="hidden" value="Gryffindor"/>
```

13.9.12. The `errors` tag

This tag renders field errors in an HTML 'span' tag. It provides access to the errors created in your controller or those that were created by any validators associated with your controller.

Let's assume we want to display all error messages for the `firstName` and `lastName` fields once we submit the form. We have a validator for instances of the `User` class called `UserValidator`.

```
public class UserValidator implements Validator {

    public boolean supports(Class candidate) {
        return User.class.isAssignableFrom(candidate);
    }

    public void validate(Object obj, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName", "required", "Field is required.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "lastName", "required", "Field is required.");
    }
}
```

The `form.jsp` would look like:

```
<form:form>
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <%-- Show errors for firstName field --%>
            <td><form:errors path="firstName" /></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <%-- Show errors for lastName field --%>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

If we submit a form with empty values in the `firstHame` and `lastName` fields, this is what the HTML would look like:

```
<form method="POST">
    <table>
        <tr>
            <td>First Name:</td>
```

```
            <td><input name="firstName" type="text" value=""/></td>
            <%-- Associated errors to firstName field displayed --%>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <%-- Associated errors to lastName field displayed --%>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form>
```

What if we want to display the entire list of errors for a given page? The example below shows that the `errors` tag also supports some basic wildcarding functionality.

- `path="*"` - displays all errors
- `path="lastName*"` - displays all errors associated with the `lastName` field

The example below will display a list of errors at the top of the page, followed by field-specific errors next to the fields:

```
<form:form>
    <form:errors path="*" cssClass="errorBox" />
    <table>
        <tr>
            <td>First Name:</td>
            <td><form:input path="firstName" /></td>
            <td><form:errors path="firstName" /></td>
        </tr>
        <tr>
            <td>Last Name:</td>
            <td><form:input path="lastName" /></td>
            <td><form:errors path="lastName"  /></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
        </tr>
    </table>
</form:form>
```

The HTML would look like:

```
<form method="POST">
    <span name="*.errors" class="errorBox">Field is required.<br/>Field is required.</span>
    <table>
        <tr>
            <td>First Name:</td>
            <td><input name="firstName" type="text" value=""/></td>
            <td><span name="firstName.errors">Field is required.</span></td>
        </tr>

        <tr>
            <td>Last Name:</td>
            <td><input name="lastName" type="text" value=""/></td>
            <td><span name="lastName.errors">Field is required.</span></td>
        </tr>
        <tr>
            <td colspan="3">
                <input type="submit" value="Save Changes" />
            </td>
```

```
        </tr>
</form>
```

13.10. Handling exceptions

Spring provides `HandlerExceptionResolvers` to ease the pain of unexpected exceptions occurring while your request is being handled by a controller which matched the request. `HandlerExceptionResolvers` somewhat resemble the exception mappings you can define in the web application descriptor `web.xml`. However, they provide a more flexible way to handle exceptions. They provide information about what handler was executing when the exception was thrown. Furthermore, a programmatic way of handling exception gives you many more options for how to respond appropriately before the request is forwarded to another URL (the same end result as when using the servlet specific exception mappings).

Besides implementing the `HandlerExceptionResolver` interface, which is only a matter of implementing the `resolveException(Exception, Handler)` method and returning a `ModelAndView`, you may also use the `SimpleMappingExceptionResolver`. This resolver enables you to take the class name of any exception that might be thrown and map it to a view name. This is functionally equivalent to the exception mapping feature from the Servlet API, but it's also possible to implement more fine grained mappings of exceptions from different handlers.

13.11. Convention over configuration

For a lot of projects, sticking to established conventions and having reasonable defaults is just what they (the projects) need... this theme of convention-over-configuration now has explicit support in Spring Web MVC. What this means is that if you establish a set of naming conventions and suchlike, you can *substantially* cut down on the amount of configuration that is required to set up handler mappings, view resolvers, `ModelAndView` instances, etc. This is a great boon with regards to rapid prototyping, and can also lend a degree of (always good-to-have) consistency across a codebase should you choose to move forward with it into production.

Tip

The Spring distribution ships with a web application that showcases the convention over configuration support described in this section. The application can be found in the `'samples/showcases/mvc-convention'` directory.

This convention over configuration support address the three core areas of MVC - namely, the models, views, and controllers.

13.11.1. The Controller - `ControllerClassNameHandlerMapping`

The `ControllerClassNameHandlerMapping` class is a `HandlerMapping` implementation that uses a convention to determine the mapping between request URLs and the `Controller` instances that are to handle those requests.

An example; consider the following (simplistic) `Controller` implementation. Take especial notice of the *name* of the class.

```
public class ViewShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // the implementation is not hugely important for this example...
```

```
    }
}
```

Here is a snippet from the attendent Spring Web MVC configuration file...

```
<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

<bean id="viewShoppingCart" class="x.y.z.ViewShoppingCartController">
    <!-- inject dependencies as required... -->
</bean>
```

The `ControllerClassNameHandlerMapping` finds all of the various handler (or `Controller`) beans defined in its application context and strips `'Controller'` off the name to define its handler mappings.

Let's look at some more examples so that the central idea becomes immediately familiar.

- `WelcomeController` maps to the `'/welcome*'` request URL
- `HomeController` maps to the `'/home*'` request URL
- `IndexController` maps to the `'/index*'` request URL
- `RegisterController` maps to the `'/register*'` request URL
- `DisplayShoppingCartController` maps to the `'/displayshoppingcart*'` request URL

  (Notice the casing - all lowercase - in the case of camel-cased `Controller` class names.)

In the case of `MultiActionController` handler classes, the mappings generated are (ever so slightly) more complex, but hopefully no less understandable. Some examples (all of the `Controller` names in this next bit are assumed to be `MultiActionController` implementations).

- `AdminController` maps to the `'/admin/*'` request URL
- `CatalogController` maps to the `'/catalog/*'` request URL

If you follow the pretty standard convention of naming your `Controller` implementations as `xxx`**`Controller`**, then the `ControllerClassNameHandlerMapping` will save you the tedium of having to firstly define and then having to maintain a potentially *looooong* `SimpleUrlHandlerMapping` (or suchlike).

The `ControllerClassNameHandlerMapping` class extends the `AbstractHandlerMapping` base class so you can define `HandlerInterceptor` instances and everything else just like you would with many other `HandlerMapping` implementations.

13.11.2. The Model - `ModelMap` (`ModelAndView`)

The `ModelMap` class is an essentially glorified `Map` that can make adding objects that are to be displayed in (or on) a `View` adhere to a common naming convention. Consider the following `Controller` implementation; notice that objects are added to the `ModelAndView` without any associated name being specified.

```
public class DisplayShoppingCartController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {

        List cartItems = // get a List of CartItem objects
        User user = // get the User doing the shopping
```

```
        ModelAndView mav = new ModelAndView("displayShoppingCart"); <-- the logical view name

        mav.addObject(cartItems); <-- look ma, no name, just the object
        mav.addObject(user); <-- and again ma!

        return mav;
    }
}
```

The `ModelAndView` class uses a `ModelMap` class that is a custom `Map` implementation that automatically generates a key for an object when an object is added to it. The strategy for determining the name for an added object is, in the case of a scalar object such as `User`, to use the short class name of the object's class. Find below some examples of the names that are generated for scalar objects put into a `ModelMap` instance.

- An `x.y.User` instance added will have the name `'user'` generated
- An `x.y.Registration` instance added will have the name `'registration'` generated
- An `x.y.Foo` instance added will have the name `'foo'` generated
- A `java.util.HashMap` instance added will have the name `'hashMap'` generated (you'll probably want to be explicit about the name in this case because `'hashMap'` is less than intuitive).
- Adding `null` will result in an `IllegalArgumentException` being thrown. If the object (or objects) that you are adding could potentially be `null`, then you will also want to be explicit about the name).

What, no automatic pluralisation?

Spring Web MVC's convention over configuration support does not support automatic pluralisation. That is to say, you cannot add a `List` of `Person` objects to a `ModelAndView` and have the generated name be 'people'.

This decision was taken after some debate, with the “Principle of Least Surprise” winning out in the end.

The strategy for generating a name after adding a `Set`, `List` or array object is to peek into the collection, take the short class name of the first object in the collection, and use that with `'List'` appended to the name. Some examples will make the semantics of name generation for collections clearer...

- An `x.y.User[]` array with one or more `x.y.User` elements added will have the name `'userList'` generated
- An `x.y.Foo[]` array with one or more `x.y.User` elements added will have the name `'fooList'` generated
- A `java.util.ArrayList` with one or more `x.y.User` elements added will have the name `'userList'` generated
- A `java.util.HashSet` with one or more `x.y.Foo` elements added will have the name `'fooList'` generated
- An **empty** `java.util.ArrayList` will not be added at all (i.e. the `addObject(..)` call will essentially be a no-op).

13.11.3. The View - `RequestToViewNameTranslator`

The `RequestToViewNameTranslator` interface is responsible for determining a logical `View` name when no such logical view name is explicitly supplied. It has just one implementation, the rather cunningly named

`DefaultRequestToViewNameTranslator` class.

The `DefaultRequestToViewNameTranslator` maps request URLs to logical view names in a fashion that is probably best explained by recourse to an example.

```
public class RegistrationController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response) {
        // process the request...
        ModelAndView mav = new ModelAndView();
        // add data as necessary to the model...
        return mav;
        // notice that no View or logical view name has been set
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
<beans>

    <!-- this bean with the well known name generates view names for us -->
    <bean id="viewNameTranslator" class="org.springframework.web.servlet.view.DefaultRequestToViewNameTranslator

    <bean class="x.y.RegistrationController">
        <!-- inject dependencies as necessary -->
    </bean>

    <!-- maps request URLs to Controller names -->
    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping"/>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/"/>
        <property name="suffix" value=".jsp"/>
    </bean>

</beans>
```

Notice how in the implementation of the `handleRequest(..)` method no `View` or logical view name is ever set on the `ModelAndView` that is returned. It is the `DefaultRequestToViewNameTranslator` that will be tasked with generating a *logical view name* from the URL of the request. In the case of the above `RegistrationController`, which is being used in conjunction with the `ControllerClassNameHandlerMapping`, a request URL of `'http://localhost/registration.html'` will result in a logical view name of `'registration'` being generated by the `DefaultRequestToViewNameTranslator`. This logical view name will then be resolved into the `'/WEB-INF/jsp/registration.jsp'` view by the `InternalResourceViewResolver` bean.

Tip

You don't even need to define a `DefaultRequestToViewNameTranslator` bean explicitly. If you are okay with the default settings of the `DefaultRequestToViewNameTranslator`, then you can rely on the fact that the Spring Web MVC `DispatcherServlet` will actually instantiate an instance of this class if one is not explicitly configured.

Of course, if you need to change the default settings, then you do need to configure your own `DefaultRequestToViewNameTranslator` bean explicitly. Please do consult the quite comprehensive Javadoc for the `DefaultRequestToViewNameTranslator` class for details of the various properties that can be configured.

13.12. Further Resources

Find below links and pointers to further resources about Spring Web MVC.

- The Spring distribution ships with a Spring Web MVC tutorial that guides the reader through building a complete Spring Web MVC-based application using a step-by-step approach. This tutorial is available in the `'docs'` directory of the Spring distribution. An online version can also be found on the Spring Framework website.
- The book entitled “Expert Spring Web MVC and WebFlow” by Seth Ladd and others (published by Apress) is an excellent hardcopy source of Spring Web MVC goodness.

Chapter 14. Integrating view technologies

14.1. Introduction

One of the areas in which Spring excels is in the separation of view technologies from the rest of the MVC framework. For example, deciding to use Velocity or XSLT in place of an existing JSP 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 13.5, “Views and resolving them” which covers the basics of how views in general are coupled to the MVC framework.

14.2. JSP & JSTL

Spring provides a couple of out-of-the-box solutions for JSP and JSTL views. Using JSP or JSTL is done using a normal view resolver defined in the `WebApplicationContext`. Furthermore, of course you need to write some JSPs that will actually render the view.

14.2.1. 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"/>
</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"/>
  <property name="prefix" value="/WEB-INF/jsp/"/>
  <property name="suffix" value=".jsp"/>
</bean>
```

The `InternalResourceBundleViewResolver` can be configured for using JSPs as described above. As a best practice, we strongly encourage placing your JSP files in a directory under the `'WEB-INF'` directory, so there can be no direct access by clients.

14.2.2. 'Plain-old' JSPs versus JSTL

When using the Java Standard Tag Library you must use a special view class, the `JstlView`, as JSTL needs

some preparation before things such as the i18N features will work.

14.2.3. 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 few tags that make things even easier. All Spring tags have *html escaping* features to enable or disable escaping of characters.

The tag library descriptor (TLD) is included in the `spring.jar` as well in the distribution itself. Further information about the individual tags can be found in the appendix entitled Appendix D, *spring.tld*.

14.3. Tiles

It is possible to integrate Tiles - just as any other view technology - in web applications using Spring. The following describes in a broad way how to do this.

NOTE: Spring 2.0 supports Tiles 1.x (a.k.a. "Struts Tiles", as shipped with Struts 1.1+). There is no out of the box support for Tiles 2 yet.

14.3.1. Dependencies

To be able to use Tiles you have to have a couple of additional dependencies included in your project. The following is the list of dependencies you need.

- `Struts version 1.1 or higher`
- `Commons BeanUtils`
- `Commons Digester`
- `Commons Logging`

These dependencies are all available in the Spring distribution.

14.3.2. 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://struts.apache.org). In Spring this is done using the `TilesConfigurer`. Have a look at the following piece of example ApplicationContext configuration:

```
<bean id="tilesConfigurer" class="org.springframework.web.servlet.view.tiles.TilesConfigurer">
  <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 definitions factory will be initialized. After that has been done, the Tiles includes in the definition files can be used as views within

your Spring web application. 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 `UrlBasedViewResolver` and the `ResourceBundleViewResolver`.

14.3.2.1. `UrlBasedViewResolver`

The `UrlBasedViewResolver` instantiates the given `viewClass` for each view it has to resolve.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass" value="org.springframework.web.servlet.view.tiles.TilesView"/>
</bean>
```

14.3.2.2. `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"/>
</bean>
```

```
...
welcomeView.class=org.springframework.web.servlet.view.tiles.TilesView
welcomeView.url=welcome (this is the name of a Tiles definition)

vetsView.class=org.springframework.web.servlet.view.tiles.TilesView
vetsView.url=vetsView (again, this is the name of a Tiles definition)

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 easily mix different view technologies.

Note that the `TilesView` class does *not* support JSTL (the JSP Standard Tag Library) out of the box. There is a separate `TilesJstlView` subclass for JSTL-enabled templates.

14.4. Velocity & FreeMarker

Velocity and FreeMarker are two templating languages that can both be used as view technologies within Spring MVC applications. The languages are quite similar and serve similar needs and so are considered together in this section. For semantic and syntactic differences between the two languages, see the FreeMarker web site.

14.4.1. Dependencies

Your web application will need to include `velocity-1.x.x.jar` or `freemarker-2.x.jar` in order to work with Velocity or FreeMarker respectively and `commons-collections.jar` needs also to be available for Velocity. Typically they are included in the `WEB-INF/lib` folder where they are 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'` directory too! The latest stable velocity, freemarker and commons collections jars are supplied with the Spring framework and can be copied from the relevant `/lib/` sub-directories. If you make use of Spring's dateToolAttribute or numberToolAttribute in your Velocity views, you will also need to include the `velocity-tools-generic-1.x.jar`

14.4.2. Context configuration

A suitable configuration is initialized by adding the relevant configurer bean definition to your `'*-servlet.xml'` as shown below:

```
<!--
  This bean sets up the Velocity environment for us based on a root path for templates.
  Optionally, a properties file can be specified for more control over the Velocity
  environment, but the defaults are pretty sane for file based template loading.
-->
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
</bean>

<!--

  View resolvers can also be configured with ResourceBundles or XML files. If you need
  different view resolving based on Locale, you have to use the resource bundle resolver.

-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".vm"/>

  <!-- if you want to use the Spring Velocity macros, set this property to true -->
  <property name="exposeSpringMacroHelpers" value="true"/>

</bean>
```

```
<!-- freemarker config -->
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
</bean>

<!--

  View resolvers can also be configured with ResourceBundles or XML files. If you need
  different view resolving based on Locale, you have to use the resource bundle resolver.

-->
<bean id="viewResolver" class="org.springframework.web.servlet.view.freemarker.FreeMarkerViewResolver">
  <property name="cache" value="true"/>
  <property name="prefix" value=""/>
  <property name="suffix" value=".ftl"/>

  <!-- if you want to use the Spring FreeMarker macros, set this property to true -->
  <property name="exposeSpringMacroHelpers" value="true"/>

</bean>
```

Note

For non web-apps add a `VelocityConfigurationFactoryBean` or a `FreeMarkerConfigurationFactoryBean` to your application context definition file.

14.4.3. Creating templates

Your templates need to be stored in the directory specified by the `*Configurer` bean shown above. This document does not cover details of creating templates for the two languages - please see their relevant websites for information. If you use the view resolvers highlighted, then the logical view names relate to the template file names in similar fashion to `InternalResourceViewResolver` for JSP's. So if your controller returns a ModelAndView object containing a view name of "welcome" then the resolvers will look for the `/WEB-INF/freemarker/welcome.ftl` or `/WEB-INF/velocity/welcome.vm` template as appropriate.

14.4.4. Advanced configuration

The basic configurations highlighted above will be suitable for most application requirements, however additional configuration options are available for when unusual or advanced requirements dictate.

14.4.4.1. velocity.properties

This file is completely optional, but if specified, contains the values that are passed to the Velocity runtime in order to configure velocity itself. Only required for advanced configurations, if you need this file, specify its location on the `VelocityConfigurer` bean definition above.

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="configLocation value="/WEB-INF/velocity.properties"/>
</bean>
```

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.

```
<bean id="velocityConfig" class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
  <property name="velocityProperties">
    <props>
      <prop key="resource.loader">file</prop>
      <prop key="file.resource.loader.class">
        org.apache.velocity.runtime.resource.loader.FileResourceLoader
      </prop>
      <prop key="file.resource.loader.path">${webapp.root}/WEB-INF/velocity</prop>
      <prop key="file.resource.loader.cache">false</prop>
    </props>
  </property>
</bean>
```

Refer to the API documentation for Spring configuration of Velocity, or the Velocity documentation for examples and definitions of the `'velocity.properties'` file itself.

14.4.4.2. FreeMarker

FreeMarker 'Settings' and 'SharedVariables' can be passed directly to the FreeMarker `Configuration` object managed by Spring by setting the appropriate bean properties on the `FreeMarkerConfigurer` bean. The `freemarkerSettings` property requires a `java.util.Properties` object and the `freemarkerVariables` property requires a `java.util.Map`.

```
<bean id="freemarkerConfig" class="org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer">
  <property name="templateLoaderPath" value="/WEB-INF/freemarker/"/>
  <property name="freemarkerVariables">
    <map>
      <entry key="xml_escape" value-ref="fmXmlEscape"/>
    </map>
  </property>
</bean>

<bean id="fmXmlEscape" class="freemarker.template.utility.XmlEscape"/>
```

See the FreeMarker documentation for details of settings and variables as they apply to the `Configuration` object.

14.4.5. Bind support and 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 values from form backing objects and to show the results of failed validations from a `Validator` in the web or business tier. From version 1.1, Spring now has support for the same functionality in both Velocity and FreeMarker, with additional convenience macros for generating form input elements themselves.

14.4.5.1. The bind macros

A standard set of macros are maintained within the `spring.jar` file for both languages, so they are always available to a suitably configured application. However they can only be used if your view sets the bean property `exposeSpringMacroHelpers` to `'true'` on your `VelocityView` / `FreeMarkerView` beans. By way of a shortcut, you can also configure this property on `VelocityViewResolver` or `FreeMarkerViewResolver` too if you happen to be using it, in which case all of your views will inherit the value from it. Note that this property is **not required** for any aspect of HTML form handling **except** where you wish to take advantage of the Spring macros. Below is an example of a view.properties file showing correct configuration of such a view for either language;

```
personFormV.class=org.springframework.web.servlet.view.velocity.VelocityView
personFormV.url=personForm.vm
personFormV.exposeSpringMacroHelpers=true
```

```
personFormF.class=org.springframework.web.servlet.view.freemarker.FreeMarkerView
personFormF.url=personForm.ftl
personFormF.exposeSpringMacroHelpers=true
```

Find below an example of a complete (albeit trivial) Spring Web MVC configuration file that exposes the Velocity macros to every (Velocity) view.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN" "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

    <bean name="helloController" class="info.wilhelms.springwebapp.SampleController">
        <property name="commandName" value="command"/>
        <property name="commandClass" value="info.wilhelms.springwebapp.Message"/>
        <property name="formView" value="foo"/>
        <property name="successView" value="banjo"/>
        <property name="bindOnNewForm" value="true"/>
        <property name="sessionForm" value="true"/>
    </bean>

    <bean id="velocityConfig"
          class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
        <property name="resourceLoaderPath" value="/WEB-INF/velocity/"/>
    </bean>

    <bean id="viewResolver" class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
        <property name="cache" value="false"/>
        <property name="prefix" value=""/>
        <property name="suffix" value=".vm"/>
        <property name="exposeSpringMacroHelpers" value="true"/>
    </bean>

    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <value>
                **/hello.htm=helloController
            </value>
        </property>
    </bean>

</beans>
```

Some of the macros defined in the Spring libraries are considered internal (private) but no such scoping exists in the macro definitions making all macros visible to calling code and user templates. The following sections

concentrate only on the macros you need to be directly calling from within your templates. If you wish to view the macro code directly, the files are called spring.vm / spring.ftl and are in the packages `org.springframework.web.servlet.view.velocity` or `org.springframework.web.servlet.view.freemarker` respectively.

14.4.5.2. simple binding

In your html forms (vm / ftl templates) that act as the 'formView' for a Spring form controller, you can use code similar to the following to bind to field values and display error messages for each input field in similar fashion to the JSP equivalent. Note that the name of the command object is "command" by default, but can be overridden in your MVC configuration by setting the 'commandName' bean property on your form controller. Example code is shown below for the `personFormV` and `personFormF` views configured earlier;

```
<!-- velocity macros are automatically available -->
<html>
...
<form action="" method="POST">
  Name:
  #springBind( "command.name" )
  <input type="text"
    name="${status.expression}"
    value="$!status.value" /><br>
  #foreach($error in $status.errorMessages) <b>$error</b> <br> #end
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

```
<!-- freemarker macros have to be imported into a namespace.  We strongly
recommend sticking to 'spring' -->
<#import "spring.ftl" as spring />
<html>
...
<form action="" method="POST">
  Name:
  <@spring.bind "command.name" />
  <input type="text"
    name="${spring.status.expression}"
    value="${spring.status.value?default("")}" /><br>
  <#list spring.status.errorMessages as error> <b>${error}</b> <br> </#list>
  <br>
  ...
  <input type="submit" value="submit"/>
</form>
...
</html>
```

`#springBind` / `<@spring.bind>` requires a 'path' argument which consists of the name of your command object (it will be 'command' unless you changed it in your FormController properties) followed by a period and the name of the field on the command object you wish to bind to. Nested fields can be used too such as "command.address.street". The `bind` macro assumes the default HTML escaping behavior specified by the ServletContext parameter `defaultHtmlEscape` in web.xml

The optional form of the macro called `#springBindEscaped` / `<@spring.bindEscaped>` takes a second argument and explicitly specifies whether HTML escaping should be used in the status error messages or values. Set to true or false as required. Additional form handling macros simplify the use of HTML escaping and these macros should be used wherever possible. They are explained in the next section.

14.4.5.3. form input generation macros

Additional convenience macros for both languages simplify both binding and form generation (including validation error display). It is never necessary to use these macros to generate form input fields, and they can be mixed and matched with simple HTML or calls direct to the spring bind macros highlighted previously.

The following table of available macros show the VTL and FTL definitions and the parameter list that each takes.

Table 14.1. Table of macro definitions

macro	VTL definition	FTL definition
message (output a string from a resource bundle based on the code parameter)	`#springMessage($code)`	`<@spring.message code/>`
messageText (output a string from a resource bundle based on the code parameter, falling back to the value of the default parameter)	`#springMessageText($code $text)`	`<@spring.messageText code, text/>`
url (prefix a relative URL with the application's context root)	`#springUrl($relativeUrl)`	`<@spring.url relativeUrl/>`
formInput (standard input field for gathering user input)	`#springFormInput($path $attributes)`	`<@spring.formInput path, attributes, fieldType/>`
formHiddenInput * (hidden input field for submitting non-user input)	`#springFormHiddenInput($path $attributes)`	`<@spring.formHiddenInput path, attributes/>`
formPasswordInput * (standard input field for gathering passwords. Note that no value will ever be populated in fields of this type)	`#springFormPasswordInput($path $attributes)`	`<@spring.formPasswordInput path, attributes/>`
formTextarea (large text field for gathering long, freeform text input)	`#springFormTextarea($path $attributes)`	`<@spring.formTextarea path, attributes/>`
formSingleSelect (drop down box of options allowing a single required value to be selected)	`#springFormSingleSelect( $path $options $attributes)`	`<@spring.formSingleSelect path, options, attributes/>`
formMultiSelect (a list box of options allowing the user to select 0 or more values)	`#springFormMultiSelect($path $options $attributes)`	`<@spring.formMultiSelect path, options, attributes/>`
formRadioButtons (a set of radio buttons allowing a single selection to be made from the available choices)	`#springFormRadioButtons($path $options $separator $attributes)`	`<@spring.formRadioButtons path, options separator, attributes/>`
formCheckboxes (a set of checkboxes allowing 0 or more values to be selected)	`#springFormCheckboxes($path $options $separator $attributes)`	`<@spring.formCheckboxes path, options, separator, attributes/>`
showErrors (simplify display of validation errors for the bound	`#springShowErrors($separator $classOrStyle)`	`<@spring.showErrors separator, classOrStyle/>`

macro	VTL definition	FTL definition
field)		

* In FTL (FreeMarker), these two macros are not actually required as you can use the normal `formInput` macro, specifying '`hidden`' or '`password`' as the value for the `fieldType` parameter.

The parameters to any of the above macros have consistent meanings:

- path: the name of the field to bind to (ie "command.name")
- options: a Map of all the available values that can be selected from in the input field. The keys to the map represent the values that will be POSTed back from the form and bound to the command object. Map objects stored against the keys are the labels displayed on the form to the user and may be different from the corresponding values posted back by the form. Usually such a map is supplied as reference data by the controller. Any Map implementation can be used depending on required behavior. For strictly sorted maps, a `SortedMap` such as a `TreeMap` with a suitable Comparator may be used and for arbitrary Maps that should return values in insertion order, use a `LinkedHashMap` or a `LinkedMap` from commons-collections.
- separator: where multiple options are available as discreet elements (radio buttons or checkboxes), the sequence of characters used to separate each one in the list (ie "
").
- attributes: an additional string of arbitrary tags or text to be included within the HTML tag itself. This string is echoed literally by the macro. For example, in a textarea field you may supply attributes as 'rows="5" cols="60"' or you could pass style information such as 'style="border:1px solid silver"'.
- classOrStyle: for the showErrors macro, the name of the CSS class that the span tag wrapping each error will use. If no information is supplied (or the value is empty) then the errors will be wrapped in tags.

Examples of the macros are outlined below some in FTL and some in VTL. Where usage differences exist between the two languages, they are explained in the notes.

14.4.5.3.1. Input Fields

```
<!-- the Name field example from above using form macros in VTL -->
...
    Name:
    #springFormInput("command.name" "")<br>
    #springShowErrors("<br>" "")<br>
```

The formInput macro takes the path parameter (command.name) and an additional attributes parameter which is empty in the example above. The macro, along with all other form generation macros, performs an implicit spring bind on the path parameter. The binding remains valid until a new bind occurs so the showErrors macro doesn't need to pass the path parameter again - it simply operates on whichever field a bind was last created for.

The showErrors macro takes a separator parameter (the characters that will be used to separate multiple errors on a given field) and also accepts a second parameter, this time a class name or style attribute. Note that FreeMarker is able to specify default values for the attributes parameter, unlike Velocity, and the two macro calls above could be expressed as follows in FTL:

```
<@spring.formInput "command.name"/>
<@spring.showErrors "<br>"/>
```

Output is shown below of the form fragment generating the name field, and displaying a validation error after

the form was submitted with no value in the field. Validation occurs through Spring's Validation framework.

The generated HTML looks like this:

```
Name:
  <input type="text" name="name" value=""
>
<br>
  <b>required</b>
<br>
<br>
```

The formTextarea macro works the same way as the formInput macro and accepts the same parameter list. Commonly, the second parameter (attributes) will be used to pass style information or rows and cols attributes for the textarea.

14.4.5.3.2. Selection Fields

Four selection field macros can be used to generate common UI value selection inputs in your HTML forms.

- formSingleSelect
- formMultiSelect
- formRadioButtons
- formCheckboxes

Each of the four macros accepts a Map of options containing the value for the form field, and the label corresponding to that value. The value and the label can be the same.

An example of radio buttons in FTL is below. The form backing object specifies a default value of 'London' for this field and so no validation is necessary. When the form is rendered, the entire list of cities to choose from is supplied as reference data in the model under the name 'cityMap'.

```
...
  Town:
  <@spring.formRadioButtons "command.address.town", cityMap, "" /><br><br>
```

This renders a line of radio buttons, one for each value in `cityMap` using the separator "". No additional attributes are supplied (the last parameter to the macro is missing). The cityMap uses the same String for each key-value pair in the map. The map's keys are what the form actually submits as POSTed request parameters, map values are the labels that the user sees. In the example above, given a list of three well known cities and a default value in the form backing object, the HTML would be

```
Town:
<input type="radio" name="address.town" value="London"

>
London
<input type="radio" name="address.town" value="Paris"
  checked="checked"
>
Paris
<input type="radio" name="address.town" value="New York"

>
New York
```

If your application expects to handle cities by internal codes for example, the map of codes would be created with suitable keys like the example below.

```
protected Map referenceData(HttpServletRequest request) throws Exception {
  Map cityMap = new LinkedHashMap();
  cityMap.put("LDN", "London");
  cityMap.put("PRS", "Paris");
  cityMap.put("NYC", "New York");

  Map m = new HashMap();
  m.put("cityMap", cityMap);
  return m;
}
```

The code would now produce output where the radio values are the relevant codes but the user still sees the more user friendly city names.

```
Town:
<input type="radio" name="address.town" value="LDN"

>
London
<input type="radio" name="address.town" value="PRS"
  checked="checked"
>
Paris
<input type="radio" name="address.town" value="NYC"

>
New York
```

14.4.5.4. HTML escaping and XHTML compliance

Default usage of the form macros above will result in HTML tags that are HTML 4.01 compliant and that use the default value for HTML escaping defined in your web.xml as used by Spring's bind support. In order to make the tags XHTML compliant or to override the default HTML escaping value, you can specify two variables in your template (or in your model where they will be visible to your templates). The advantage of specifying them in the templates is that they can be changed to different values later in the template processing to provide different behavior for different fields in your form.

To switch to XHTML compliance for your tags, specify a value of 'true' for a model/context variable named xhtmlCompliant:

```
## for Velocity..
#set($springXhtmlCompliant = true)

<#-- for FreeMarker -->
<#assign xhtmlCompliant = true in spring>
```

Any tags generated by the Spring macros will now be XHTML compliant after processing this directive.

In similar fashion, HTML escaping can be specified per field:

```
<#-- until this point, default HTML escaping is used -->

<#assign htmlEscape = true in spring>
<#-- next field will use HTML escaping -->
<@spring.formInput "command.name" />

<#assign htmlEscape = false in spring>
<#-- all future fields will be bound with HTML escaping off -->
```

14.5. XSLT

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 Web MVC application.

14.5.1. 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 the section entitled Section 13.3, “Controllers” for details of Spring Web MVC's `Controller` interface. The XSLT view will turn the list of words into a simple XML document ready for transformation.

14.5.1.1. 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 encapsulates our word generation logic.

14.5.1.2. 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 request,
    HttpServletResponse response) 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.

14.5.1.3. Convert the model data to XML

In order to create a DOM document from our list of words or any other model data, we must subclass the

(provided) `org.springframework.web.servlet.view.xslt.AbstractXsltView` class. In doing so, we must also typically implement the abstract method `createXsltSource(..)` method. 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:

```
package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

    protected Source createXsltSource(Map model, String rootName, HttpServletRequest
        request, HttpServletResponse response) throws Exception {

        Document document = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();
        Element root = document.createElement(rootName);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element wordNode = document.createElement("word");
            Text textNode = document.createTextNode(nextWord);
            wordNode.appendChild(textNode);
            root.appendChild(wordNode);
        }
        return new DOMSource(root);
    }

}
```

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 `getParameters()` method of the `AbstractXsltView` class and return a `Map` of the name/value pairs. If your parameters need to derive information from the current request, you can (from Spring version 1.1) override the `getParameters(HttpServletRequest request)` method instead.

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 `createXsltSource(..)` method(s) to get such support. See the Javadocs for the `org.springframework.web.servlet.view.xslt.FormatHelper` class.

14.5.1.4. 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 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 `createXsltSource(..)` method(s).

14.5.1.5. Document transformation

Finally, we have the XSLT code used for transforming the above document. As shown in the above

'views.properties' file, the stylesheet is called 'home.xslt' and it lives in the war file in the 'WEB-INF/xsl' directory.

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

    <xsl:output method="html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>
                <h1>My First Words</h1>
                <xsl:apply-templates/>
            </body>
        </html>
    </xsl:template>

    <xsl:template match="word">
        <xsl:value-of select="."/><br/>
    </xsl:template>

</xsl:stylesheet>
```

14.5.2. 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.

14.6. Document views (PDF/Excel)

14.6.1. 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 correct content type to (hopefully) enable the client PC to run their 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.

14.6.2. 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).

14.6.2.1. 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

14.6.2.2. 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!

14.6.2.3. Subclassing for Excel views

Exactly as we did for the XSLT example, we'll subclass suitable abstract classes in order to implement custom behavior in generating our output documents. For Excel, this involves writing a subclass of `org.springframework.web.servlet.view.document.AbstractExcelView` (for Excel files generated by POI) or `org.springframework.web.servlet.view.document.AbstractJExcelView` (for JExcelApi-generated Excel files). and implementing the `buildExcelDocument`

Here's the complete listing for our POI 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));

        }
    }
}
```

And this a view generating the same Excel file, now using JExcelApi:

```
package excel;

// imports omitted for brevity

public class HomePage extends AbstractExcelView {

    protected void buildExcelDocument(Map model,
        WritableWorkbook wb,
        HttpServletRequest request,
        HttpServletResponse response)
    throws Exception {

        WritableSheet sheet = wb.createSheet("Spring");

        sheet.addCell(new Label(0, 0, "Spring-Excel test");

        List words  = (List)model.get("wordList");
        for (int i = -; i < words.size(); i++) {
            sheet.addCell(new Label(2+i, 0, (String)words.get(i));
        }
    }
}
```

Note the differences between the APIs. We've found that the JExcelApi is somewhat more intuitive and furthermore, JExcelApi has a bit better image-handling capabilities. There have been memory problems with large Excel file when using JExcelApi however.

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 automatically when you request the same page as before.

14.6.2.4. 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.

14.7. JasperReports

JasperReports (http://jasperreports.sourceforge.net) is a powerful open-source reporting engine that supports the creation of report designs using an easily understood XML file format. JasperReports is capable of rendering reports output into four different formats: CSV, Excel, HTML and PDF.

14.7.1. Dependencies

Your application will need to include the latest release of JasperReports, which at the time of writing was 0.6.1. JasperReports itself depends on the following projects:

- BeanShell
- Commons BeanUtils
- Commons Collections
- Commons Digester
- Commons Logging
- iText
- POI

JasperReports also requires a JAXP compliant XML parser.

14.7.2. Configuration

To configure JasperReports views in your Spring container configuration you need to define a `ViewResolver` to map view names to the appropriate view class depending on which format you want your report rendered in.

14.7.2.1. Configuring the `ViewResolver`

Typically, you will use the `ResourceBundleViewResolver` to map view names to view classes and files in a properties file.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>
```

Here we've configured an instance of the `ResourceBundleViewResolver` class that will look for view mappings in the resource bundle with base name `views`. (The content of this file is described in the next section.)

14.7.2.2. Configuring the `View`s

The Spring Framework contains five different `View` implementations for JasperReports, four of which correspond to one of the four output formats supported by JasperReports, and one that allows for the format to be determined at runtime:

Table 14.2. JasperReports `View` classes

Class Name	Render Format
`JasperReportsCsvView`	CSV
`JasperReportsHtmlView`	HTML
`JasperReportsPdfView`	PDF
`JasperReportsXlsView`	Microsoft Excel
`JasperReportsMultiFormatView`	The view is decided upon at runtime

Mapping one of these classes to a view name and a report file is a matter of adding the appropriate entries into the resource bundle configured in the previous section as shown here:

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
```

Here you can see that the view with name `simpleReport` is mapped to the `JasperReportsPdfView` class, causing the output of this report to be rendered in PDF format. The `url` property of the view is set to the location of the underlying report file.

14.7.2.3. About Report Files

JasperReports has two distinct types of report file: the design file, which has a `.jrxml` extension, and the compiled report file, which has a `.jasper` extension. Typically, you use the JasperReports Ant task to compile your `.jrxml` design file into a `.jasper` file before deploying it into your application. With the Spring Framework you can map either of these files to your report file and the framework will take care of compiling the `.jrxml` file on the fly for you. You should note that after a `.jrxml` file is compiled by the Spring Framework, the compiled report is cached for the lifetime of the application. To make changes to the file you will need to restart your application.

14.7.2.4. Using `JasperReportsMultiFormatView`

The `JasperReportsMultiFormatView` allows for report format to be specified at runtime. The actual rendering of the report is delegated to one of the other JasperReports view classes - the `JasperReportsMultiFormatView` class simply adds a wrapper layer that allows for the exact implementation to be specified at runtime.

The `JasperReportsMultiFormatView` class introduces two concepts: the format key and the discriminator key.

The `JasperReportsMultiFormatView` class uses the mapping key to lookup the actual view implementation class and uses the format key to lookup up the mapping key. From a coding perspective you add an entry to your model with the formay key as the key and the mapping key as the value, for example:

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

  String uri = request.getRequestURI();
  String format = uri.substring(uri.lastIndexOf(".") + 1);

  Map model = getModel();
  model.put("format", format);

  return new ModelAndView("simpleReportMulti", model);
}
```

In this example, the mapping key is determined from the extension of the request URI and is added to the model under the default format key: `format`. If you wish to use a different format key then you can configure this using the `formatKey` property of the `JasperReportsMultiFormatView` class.

By default the following mapping key mappings are configured in `JasperReportsMultiFormatView`:

Table 14.3. `JasperReportsMultiFormatView` Default Mapping Key Mappings

Mapping Key	View Class
csv	`JasperReportsCsvView`
html	`JasperReportsHtmlView`
pdf	`JasperReportsPdfView`
xls	`JasperReportsXlsView`

So in the example above a request to URI /foo/myReport.pdf would be mapped to the `JasperReportsPdfView` class. You can override the mapping key to view class mappings using the `formatMappings` property of `JasperReportsMultiFormatView`.

14.7.3. Populating the `ModelAndView`

In order to render your report correctly in the format you have chosen, you must supply Spring with all of the data needed to populate your report. For JasperReports this means you must pass in all report parameters along with the report datasource. Report parameters are simple name/value pairs and can be added be to the `Map` for your model as you would add any name/value pair.

When adding the datasource to the model you have two approaches to choose from. The first approach is to add an instance of `JRDataSource` or a `Collection` type to the model `Map` under any arbitrary key. Spring will then locate this object in the model and treat it as the report datasource. For example, you may populate your model like so:

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  model.put("myBeanData", beanData);
  return model;
}
```

The second approach is to add the instance of `JRDataSource` or `Collection` under a specific key and then

configure this key using the `reportDataKey` property of the view class. In both cases Spring will instances of `Collection` in a `JRBeanCollectionDataSource` instance. For example:

```
private Map getModel() {
  Map model = new HashMap();
  Collection beanData = getBeanData();
  Collection someData = getSomeData();
  model.put("myBeanData", beanData);
  model.put("someData", someData);
  return model;
}
```

Here you can see that two `Collection` instances are being added to the model. To ensure that the correct one is used, we simply modify our view configuration as appropriate:

```
simpleReport.class=org.springframework.web.servlet.view.jasperreports.JasperReportsPdfView
simpleReport.url=/WEB-INF/reports/DataSourceReport.jasper
simpleReport.reportDataKey=myBeanData
```

Be aware that when using the first approach, Spring will use the first instance of `JRDataSource` or `Collection` that it encounters. If you need to place multiple instances of `JRDataSource` or `Collection` into the model then you need to use the second approach.

14.7.4. Working with Sub-Reports

JasperReports provides support for embedded sub-reports within your master report files. There are a wide variety of mechanisms for including sub-reports in your report files. The easiest way is to hard code the report path and the SQL query for the sub report into your design files. The drawback of this approach is obvious - the values are hard-coded into your report files reducing reusability and making it harder to modify and update report designs. To overcome this you can configure sub-reports declaratively and you can include additional data for these sub-reports directly from your controllers.

14.7.4.1. Configuring Sub-Report Files

To control which sub-report files are included in a master report using Spring, your report file must be configured to accept sub-reports from an external source. To do this you declare a parameter in your report file like so:

```
<parameter name="ProductsSubReport" class="net.sf.jasperreports.engine.JasperReport"/>
```

Then, you define your sub-report to use this sub-report parameter:

```
<subreport>
    <reportElement isPrintRepeatedValues="false" x="5" y="25" width="325"
        height="20" isRemoveLineWhenBlank="true" backcolor="#ffcc99"/>
    <subreportParameter name="City">
        <subreportParameterExpression><![CDATA[$F{city}]]></subreportParameterExpression>
    </subreportParameter>
    <dataSourceExpression><![CDATA[$P{SubReportData}]]></dataSourceExpression>
    <subreportExpression class="net.sf.jasperreports.engine.JasperReport">
                <![CDATA[$P{ProductsSubReport}]]></subreportExpression>
</subreport>
```

This defines a master report file that expects the sub-report to be passed in as an instance of `net.sf.jasperreports.engine.JasperReports` under the parameter `ProductsSubReport`. When configuring your Jasper view class, you can instruct Spring to load a report file and pass into the JasperReports engine as a sub-report using the `subReportUrls` property:

```
<property name="subReportUrls">
    <map>
        <entry key="ProductsSubReport" value="/WEB-INF/reports/subReportChild.jrxml"/>
    </map>
</property>
```

Here, the key of the `Map` corresponds to the name of the sub-report parameter in th report design file, and the entry is the URL of the report file. Spring will load this report file, compiling it if necessary, and will pass into the JasperReports engine under the given key.

14.7.4.2. Configuring Sub-Report Data Sources

This step is entirely optional when using Spring configure your sub-reports. If you wish, you can still configure the data source for your sub-reports using static queries. However, if you want Spring to convert data returned in your `ModelAndView` into instances of `JRDataSource` then you need to specify which of the parameters in your `ModelAndView` Spring should convert. To do this configure the list of parameter names using the `subReportDataKeys` property of the your chosen view class:

```
<property name="subReportDataKeys"
    value="SubReportData"/>
```

Here, the key you supply MUST correspond to both the key used in your `ModelAndView` and the key used in your report design file.

14.7.5. Configuring Exporter Parameters

If you have special requirements for exporter configuration - perhaps you want a specific page size for your PDF report, then you can configure these exporter parameters declaratively in your Spring configuration file using the `exporterParameters` property of the view class. The `exporterParameters` property is typed as `Map` and in your configuration the key of an entry should be the fully-qualified name of a static field that contains the exporter parameter definition and the value of an entry should be the value you want to assign to the parameter. An example of this is shown below:

```
<bean id="htmlReport" class="org.springframework.web.servlet.view.jasperreports.JasperReportsHtmlView">
  <property name="url" value="/WEB-INF/reports/simpleReport.jrxml"/>
  <property name="exporterParameters">
    <map>
      <entry key="net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER">
        <value>Footer by Spring!
          &lt;/td&gt;&lt;td width="50%"&gt;&amp;nbsp; &lt;/td&gt;&lt;/tr&gt;
          &lt;/table&gt;&lt;/body&gt;&lt;/html&gt;
        </value>
      </entry>
    </map>
  </property>
</bean>
```

Here you can see that the `JasperReportsHtmlView` is being configured with an exporter parameter for `net.sf.jasperreports.engine.export.JRHtmlExporterParameter.HTML_FOOTER` which will output a footer in the resulting HTML.

Chapter 15. Integrating with other web frameworks

15.1. Introduction

This chapter details Spring's integration with third party web frameworks such as Struts, JSF, Tapestry, and WebWork.

Spring WebFlow

Spring Web Flow (SWF) aims to be the best solution for the management of web application page flow.

SWF integrates with existing frameworks like Spring MVC, Struts, and JSF, in both servlet and portlet environments. If you have a business process (or processes) that would benefit from a conversational model as opposed to a purely request model, then SWF may be the solution.

SWF allows you to capture logical page flows as self-contained modules that are reusable in different situations, and as such is ideal for building web application modules that guide the user through controlled navigations that drive business processes.

For more information about SWF, consult the Spring WebFlow site.

One of the core value propositions of the Spring Framework is that of enabling *choice*. In a general sense, Spring does not force one to use or buy into any particular architecture, technology, or methodology (although it certainly recommends some over others). This freedom to pick and choose the architecture, technology, or methodology that is most relevant to a developer and his or her development team is arguably most evident in the web area, where Spring provides its own web framework (SpringMVC), while at the same time providing integration with a number of popular third party web frameworks. This allows one to continue to leverage any and all of the skills one may have acquired in a particular web framework such as Struts, while at the same time being able to enjoy the benefits afforded by Spring in other areas such as data access, declarative transaction management, and flexible configuration and application assembly.

Having dispensed with the woolly sales patter (c.f. the previous paragraph), the remainder of this chapter will concentrate upon the meaty details of integrating your favourite web framework with Spring. One thing that is often commented upon by developers coming to Java from other languages is the seeming super-abundance of web frameworks available in Java... there are indeed a great number of web frameworks in the Java space; in fact there are far too many to cover with any semblance of detail in a single chapter. This chapter thus picks four of the more popular web frameworks in Java, starting with the Spring configuration that is common to all of the supported web frameworks, and then detailing the specific integration options for each supported web framework.

Please note that this chapter does not attempt to explain how to use any of the supported web frameworks. For example, if you want to use Struts for the presentation layer of your web application, the assumption is that you are already familiar with Struts. If you need further details about any of the supported web frameworks themselves, please do consult the section entitled Section 15.7, “Further Resources” at the end of this chapter.

15.2. Common configuration

Before diving into the integration specifics of each supported web framework, let us first take a look at the Spring configuration that *not* specific to any one web framework. (This section is equally applicable to Spring's

own web framework, SpringMVC.)

One of the concepts (for want of a better word) espoused by (Spring's) lightweight application model is that of a layered architecture. Remember that in a 'classic' layered architecture, the web layer is but one of many layers... it serves as one of the entry points into a server side application, and it delegates to service objects (facades) defined in a service layer to satisfy business specific (and presentation-technology agnostic) use cases. In Spring, these service objects, any other business-specific objects, data access objects, etc. exist in a distinct 'business context', which contains *no* web or presentation layer objects (presentation objects such as Spring MVC controllers are typically configured in a distinct 'presentation context'). This section details how one configures a Spring container (a `WebApplicationContext`) that contains all of the 'business beans' in one's application.

Onto specifics... all that one need do is to declare a `ContextLoaderListener` in the standard J2EE servlet `web.xml` file of one's web application, and add a `contextConfigLocation` <context-param/> section (in the same file) that defines which set of Spring XML cpnfiguration files to load.

Find below the <listener/> configuration:

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

Note

Listeners were added to the Servlet API in version 2.3. If you have a Servlet 2.2 container, you can use the `ContextLoaderServlet` to achieve this same functionality.

Find below the <context-param/> configuration:

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

If you don't specify the `contextConfigLocation` context parameter, the `ContextLoaderListener` will look for a file called `/WEB-INF/applicationContext.xml` to load. Once the context files are loaded, Spring creates a `WebApplicationContext` object based on the bean definitions and stores it in the ServletContext of one's web application.

All Java web frameworks are built on top of the Servlet API, and so one can use the following code snippet to get access to this 'business context' ApplicationContext created by the `ContextLoaderListener`.

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

The `WebApplicationContextUtils` class is for convenience, so you don't have to remember the name of the ServletContext attribute. Its *getWebApplicationContext()* method will return `null` if an object doesn't exist under the `WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE` key. Rather than risk getting `NullPointerExceptions` in your application, it's better to use the `getRequiredWebApplicationContext()` method. This method throws an exception when the ApplicationContext is missing.

Once you have a reference to the `WebApplicationContext`, you can retrieve beans by their name or type. Most developers retrieve beans by name, then cast them to one of their implemented interfaces.

Fortunately, most of the frameworks in this section have simpler ways of looking up beans. Not only do they

make it easy to get beans from a Spring container, but they also allow you to use dependency injection on their controllers. Each web framework section has more detail on its specific integration strategies.

15.3. JavaServer Faces

JavaServer Faces (JSF) is an increasingly popular component-based, event-driven web framework. The key class in Spring's JSF integration is the `DelegatingVariableResolver` class.

15.3.1. DelegatingVariableResolver

The easiest way to integrate one's Spring middle-tier with one's JSF web layer is to use the `DelegatingVariableResolver` class. To configure this variable resolver in one's application, one will need to edit one's *faces-context.xml* file. After the opening `<faces-config/>` element, add an `<application/>` element and a `<variable-resolver/>` element within it. The value of the variable resolver should reference Spring's `DelegatingVariableResolver`; for example:

```
<faces-config>
  <application>
    <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver</variable-resolver>
      <locale-config>
        <default-locale>en</default-locale>
        <supported-locale>en</supported-locale>
        <supported-locale>es</supported-locale>
      </locale-config>
      <message-bundle>messages</message-bundle>
    </application>
</faces-config>
```

The `DelegatingVariableResolver` will first delegate value lookups to the default resolver of the underlying JSF implementation, and then to Spring's 'business context' `WebApplicationContext`. This allows one to easily inject dependencies into one's JSF-managed beans.

Managed beans are defined in one's `faces-config.xml` file. Find below an example where `#{userManager}` is a bean that is retrieved from the Spring 'business context'.

```
<managed-bean>
  <managed-bean-name>userList</managed-bean-name>
        <managed-bean-class>com.whatever.jsf.UserList</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>userManager</property-name>
    <value>#{userManager}</value>
  </managed-property>
</managed-bean>
```

15.3.2. FacesContextUtils

A custom `VariableResolver` works well when mapping one's properties to beans in *faces-config.xml*, but at times one may need to grab a bean explicitly. The `FacesContextUtils` class makes this easy. It is similar to `WebApplicationContextUtils`, except that it takes a `FacesContext` parameter rather than a ServletContext parameter.

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

The `DelegatingVariableResolver` is the recommended strategy for integrating JSF and Spring. If one is

looking for more robust integration features, one might want take a look at the JSF-Spring project.

15.4. Struts

Struts is the *de facto* web framework for Java applications, mainly because it was one of the first to be released (June 2001). Invented by Craig McClanahan, Struts is an open source project hosted by the Apache Software Foundation. At the time, it greatly simplified the JSP/Servlet programming paradigm and won over many developers who were using proprietary frameworks. It simplified the programming model, it was open source (and thus free as in beer), and it had a large community, which allowed the project to grow and become popular among Java web developers.

To integrate your Struts application with Spring, you have two options:

- Configure Spring to manage your Actions as beans, using the `ContextLoaderPlugin`, and set their dependencies in a Spring context file.
- Subclass Spring's `ActionSupport` classes and grab your Spring-managed beans explicitly using a *getWebApplicationContext()* method.

15.4.1. ContextLoaderPlugin

The `ContextLoaderPlugin` is a Struts 1.1+ plug-in that loads a Spring context file for the Struts `ActionServlet`. This context refers to the root `WebApplicationContext` (loaded by the `ContextLoaderListener`) as its parent. The default name of the context file is the name of the mapped servlet, plus *-servlet.xml*. If `ActionServlet` is defined in web.xml as `<servlet-name>action</servlet-name>`, the default is */WEB-INF/action-servlet.xml*.

To configure this plug-in, add the following XML to the plug-ins section near the bottom of your *struts-config.xml* file:

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn"/>
```

The location of the context configuration files can be customized using the '`contextConfigLocation`' property.

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
      value="/WEB-INF/action-servlet.xml,/WEB-INF/applicationContext.xml"/>
</plug-in>
```

It is possible to use this plugin to load all your context files, which can be useful when using testing tools like StrutsTestCase. StrutsTestCase's `MockStrutsTestCase` won't initialize Listeners on startup so putting all your context files in the plugin is a workaround. (A bug has been filed for this issue, but has been closed as 'Wont Fix').

After configuring this plug-in in *struts-config.xml*, you can configure your `Action` to be managed by Spring. Spring (1.1.3+) provides two ways to do this:

- Override Struts' default `RequestProcessor` with Spring's `DelegatingRequestProcessor`.
- Use the `DelegatingActionProxy` class in the `type` attribute of your `<action-mapping>`.

Both of these methods allow you to manage your Actions and their dependencies in the *action-servlet.xml* file.

The bridge between the Action in *struts-config.xml* and *action-servlet.xml* is built with the action-mapping's "path" and the bean's "name". If you have the following in your *struts-config.xml* file:

```
<action path="/users" .../>
```

You must define that Action's bean with the "/users" name in *action-servlet.xml*:

```
<bean name="/users" .../>
```

15.4.1.1. DelegatingRequestProcessor

To configure the `DelegatingRequestProcessor` in your *struts-config.xml* file, override the "processorClass" property in the <controller> element. These lines follow the <action-mapping> element.

```
<controller>
  <set-property property="processorClass"
      value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>
```

After adding this setting, your Action will automatically be looked up in Spring's context file, no matter what the type. In fact, you don't even need to specify a type. Both of the following snippets will work:

```
<action path="/user" type="com.whatever.struts.UserAction"/>
<action path="/user"/>
```

If you're using Struts' *modules* feature, your bean names must contain the module prefix. For example, an action defined as `<action path="/user"/>` with module prefix "admin" requires a bean name with `<bean name="/admin/user"/>`.

Note

If you are using Tiles in your Struts application, you must configure your <controller> with the `DelegatingTilesRequestProcessor` instead.

15.4.1.2. DelegatingActionProxy

If you have a custom `RequestProcessor` and can't use the `DelegatingRequestProcessor` or `DelegatingTilesRequestProcessor` approaches, you can use the `DelegatingActionProxy` as the type in your action-mapping.

```
<action path="/user" type="org.springframework.web.struts.DelegatingActionProxy"
    name="userForm" scope="request" validate="false" parameter="method">
  <forward name="list" path="/userList.jsp"/>
  <forward name="edit" path="/userForm.jsp"/>
</action>
```

The bean definition in *action-servlet.xml* remains the same, whether you use a custom `RequestProcessor` or the `DelegatingActionProxy`.

If you define your `Action` in a context file, the full feature set of Spring's bean container will be available for it: dependency injection as well as the option to instantiate a new `Action` instance for each request. To activate the latter, add *scope="prototype"* to your Action's bean definition.

```
<bean name="/user" scope="prototype" autowire="byName"
    class="org.example.web.UserAction"/>
```

15.4.2. ActionSupport Classes

As previously mentioned, you can retrieve the `WebApplicationContext` from the ServletContext using the `WebApplicationContextUtils` class. An easier way is to extend Spring's `Action` classes for Struts. For example, instead of subclassing Struts' `Action` class, you can subclass Spring's `ActionSupport` class.

The `ActionSupport` class provides additional convenience methods, like *getWebApplicationContext()*. Below is an example of how you might use this in an Action:

```
public class UserAction extends DispatchActionSupport {

    public ActionForward execute(ActionMapping mapping,
                                 ActionForm form,
                                 HttpServletRequest request,
                                 HttpServletResponse response) throws Exception {
        if (log.isDebugEnabled()) {
            log.debug("entering 'delete' method...");
        }
        WebApplicationContext ctx = getWebApplicationContext();
        UserManager mgr = (UserManager) ctx.getBean("userManager");
        // talk to manager for business logic
        return mapping.findForward("success");
    }
}
```

Spring includes subclasses for all of the standard Struts Actions - the Spring versions merely have *Support* appended to the name:

- `ActionSupport`,
- `DispatchActionSupport`,
- `LookupDispatchActionSupport` and
- `MappingDispatchActionSupport`.

The recommended strategy is to use the approach that best suits your project. Subclassing makes your code more readable, and you know exactly how your dependencies are resolved. However, using the `ContextLoaderPlugin` allow you to easily add new dependencies in your context XML file. Either way, Spring provides some nice options for integrating the two frameworks.

15.5. Tapestry

From the Tapestry homepage...
“ *Tapestry is an open-source framework for creating dynamic, robust, highly scalable web applications in Java. Tapestry complements and builds upon the standard Java Servlet API, and so it works in any servlet container or application server.* ”

While Spring has its own powerful web layer, there are a number of unique advantages to building a J2EE application using a combination of Tapestry for the web user interface and the Spring container for the lower layers. This section of the web integration chapter attempts to detail a few best practices for combining these two frameworks.

A *typical* layered J2EE application built with Tapestry and Spring will consist of a top user interface (UI) layer built with Tapestry, and a number of lower layers, all wired together by one or more Spring containers. Tapestry's own reference documentation contains the following snippet of best practice advice. (Text that I the author have added is contained within `[]` brackets.)
“ *A very succesful design pattern in Tapestry is to keep pages and components very simple, and **delegate** as much logic as possible out to HiveMind [or Spring, or whatever] services. Listener methods should ideally do*

little more than marshall together the correct information and pass it over to a service. ”

The key question then is... how does one supply Tapestry pages with collaborating services? The answer, ideally, is that one would want to dependency inject those services directly into one's Tapestry pages. In Tapestry, one can effect this dependency injection by a variety of means... this section is only going to enumerate the dependency injection means afforded by Spring. The real beauty of the rest of this Spring-Tapestry integration is that the elegant and flexible design of Tapestry itself makes doing this dependency injection of Spring-managed beans a cinch. (Another nice thing is that this Spring-Tapestry itegration code was written - and continues to be maintained - by the Tapestry creator Howard M. Lewis Ship, so hats off to him for what is really some silky smooth integration).

15.5.1. Injecting Spring-managed beans

Assume we have the following simple Spring container definition (in the ubiquitous XML format):

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    <!-- the DataSource -->
    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName" value="java:DefaultDS"/>
    </bean>

    <bean id="hibSessionFactory"
          class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
    </bean>

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

    <bean id="mapper"
          class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
        <property name="sessionFactory" ref="hibSessionFactory"/>
    </bean>

    <!-- (transactional) AuthenticationService -->
    <bean id="authenticationService"
          class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="target">
            <bean class="com.whatever.services.service.user.AuthenticationServiceImpl">
                <property name="mapper" ref="mapper"/>
            </bean>
        </property>
        <property name="proxyInterfacesOnly" value="true"/>
        <property name="transactionAttributes">
            <value>
                *=PROPAGATION_REQUIRED
            </value>
        </property>
    </bean>

    <!-- (transactional) UserService -->
    <bean id="userService"
          class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="target">
             <bean class="com.whatever.services.service.user.UserServiceImpl">
                 <property name="mapper" ref="mapper"/>
             </bean>
        </property>
        <property name="proxyInterfacesOnly" value="true"/>
        <property name="transactionAttributes">
            <value>
                *=PROPAGATION_REQUIRED
            </value>
        </property>
    </bean>
```

```
</beans>
```

Inside the Tapestry application, the above bean definitions need to be loaded into a Spring container, and any relevant Tapestry pages need to be supplied (injected) with the `authenticationService` and `userService` beans, which implement the `AuthenticationService` and `UserService` interfaces, respectively.

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 = (UserService) appContext.getBean("userService");
... some code which uses UserService
```

This mechanism does work... having said that, 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 IoC principle; 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.

Note

This next section is applicable to versions of Tapestry < 4.0. if you are using Tapestry version 4.0+, please consult the section entitled Section 15.5.1.4, “Dependency Injecting Spring Beans into Tapestry pages - Tapestry 4.0+ style”.

15.5.1.1. Dependency Injecting Spring Beans into Tapestry pages

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>
```

15.5.1.2. 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>
```

15.5.1.3. 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 (in order to be able to access the properties).

```
// 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 the sake of 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 successfully 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 the cookie that identifies a user **/
    private static final String COOKIE_NAME = Login.class.getName() + ".username";
    private final static int ONE_WEEK = 7 * 24 * 60 * 60;

    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();

    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 occurred.
        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);
        }

        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));
        }
    }
}
```

15.5.1.4. Dependency Injecting Spring Beans into Tapestry pages - Tapestry 4.0+ style

Effecting the dependency injection of Spring-managed beans into Tapestry pages in Tapestry version 4.0+ is *so* much simpler. All that is needed is a single add-on library, and some (small) amount of (essentially boilerplate) configuration. Simply package and deploy this library with the (any of the) other libraries required by your web application (typically in `WEB-INF/lib`).

You will then need to create and expose the Spring container using the method detailed previously. You can then inject Spring-managed beans into Tapestry very easily; if we are using Java 5, consider the `Login` page from above: we simply need to annotate the appropriate getter methods in order to dependency inject the Spring-managed `userService` and `authenticationService` objects (lots of the class definition has been elided for clarity)...

```
package com.whatever.web.xportal.pages;

public abstract class Login extends BasePage implements ErrorProperty, PageRenderListener {

    @InjectObject("spring:userService")
    public abstract UserService getUserService();

    @InjectObject("spring:authenticationService")
    public abstract AuthenticationService getAuthenticationService();

}
```

We are almost done... all that remains is the HiveMind configuration that exposes the Spring container stored in the `ServletContext` as a HiveMind service; for example:

```
<?xml version="1.0"?>
<module id="com.javaforge.tapestry.spring" version="0.1.1">

    <service-point id="SpringApplicationInitializer"
        interface="org.apache.tapestry.services.ApplicationInitializer"
        visibility="private">
        <invoke-factory>
            <construct class="com.javaforge.tapestry.spring.SpringApplicationInitializer">
                <set-object property="beanFactoryHolder"
                    value="service:hivemind.lib.DefaultSpringBeanFactoryHolder" />
            </construct>
        </invoke-factory>
    </service-point>

    <!-- Hook the Spring setup into the overall application initialization. -->
    <contribution
        configuration-id="tapestry.init.ApplicationInitializers">
        <command id="spring-context"
            object="service:SpringApplicationInitializer" />
    </contribution>

</module>
```

If you are using Java 5 (and thus have access to annotations), then that really is it.

If you are not using Java 5, then one obviously doesn't annotate one's Tapestry page classes with annotations; instead, one simply uses good old fashioned XML to declare the dependency injection; for example, inside the `.page` or `.jwc` file for the `Login` page (or component):

```
<inject property="userService" object="spring:userService"/>
<inject property="authenticationService" object="spring:authenticationService"/>
```

In this example, we've managed to allow service beans defined in a Spring container to be provided to the Tapestry 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.

15.6. WebWork

From the WebWork homepage...
“ WebWork is a Java web-application development framework. It is built specifically with developer productivity and code simplicity in mind, providing robust support for building reusable UI templates, such as form controls, UI themes, internationalization, dynamic form parameter mapping to JavaBeans, robust client and server side validation, and much more. ”

WebWork is (in the opinion of this author) a very clean, elegant web framework. Its architecture and key concepts are not only very easy to understand, it has a rich tag library, nicely decoupled validation, and it is (again, in the opinion of this author) quite easy to be productive in next to no time at all (the documentation and tutorials are pretty good too).

One of the key enablers in WebWork's technology stack is an IoC container to manage Webwork Actions, handle the "wiring" of business objects, etc. Previous to WebWork version 2.2, WebWork used it's own proprietary IoC container (and provided integration points so that one could integrate an IoC container such as Springs into the mix). However, as of WebWork version 2.2, the default IoC container that is used within WebWork *is* Spring. This is obviously great news if one is a Spring developer, because it means that one is immediately familiar with the basics of IoC configuration, idioms and suchlike within WebWork.

Now in the interests of adhering to the DRY (Dont Repeat Yourself) principle, it would be foolish to writeup the Spring-WebWork integration in light of the fact that the WebWork team have already written such a writeup. Please do consult the Spring-WebWork integration page on the WebWork wiki for the full lowdown.

Note that the Spring-WebWork integration code was developed (and continues to be maintained and improved) by the WebWork developers themselves, so in the first instance please do refer to the WebWork site and forums if you are having issues with the integration. Do feel free to post comments and queries regarding the Spring-WebWork integration on the Spring support forums too.

15.7. Further Resources

Find below links to further resources about the various web frameworks described in this chapter.

- The Struts homepage
- The JSF homepage
- The Tapestry homepage
- The WebWork homepage

Find below some further web framework-related resources that you may find personally enriching.

- The StrutsTI project wiki

Chapter 16. Portlet MVC Framework

16.1. Introduction

JSR-168 The Java Portlet Specification

For more general information about portlet development, please review a whitepaper from Sun entitled "Introduction to JSR 168", and of course the JSR-168 Specification itself.

In addition to supporting conventional (servlet-based) Web development, Spring also supports JSR-168 Portlet development. As much as possible, the Portlet MVC framework is a mirror image of the Web MVC framework, and also uses the same underlying view abstractions and integration technology. So, be sure to review the chapters entitled Chapter 13, *Web MVC framework* and Chapter 14, *Integrating view technologies* before continuing with this chapter.

Note

Bear in mind that while the concepts of Spring MVC are the same in Spring Portlet MVC, there are some notable differences created by the unique workflow of JSR-168 portlets.

The main way in which portlet workflow differs from servlet workflow is that the request to the portlet can have two distinct phases: the action phase and the render phase. The action phase is executed only once and is where any 'backend' changes or actions occur, such as making changes in a database. The render phase then produces what is displayed to the user each time the display is refreshed. The critical point here is that for a single overall request, the action phase is executed only once, but the render phase may be executed multiple times. This provides (and requires) a clean separation between the activities that modify the persistent state of your system and the activities that generate what is displayed to the user.

Spring WebFlow

Spring Web Flow (SWF) aims to be the best solution for the management of web application page flow.

SWF integrates with existing frameworks like Spring MVC, Struts, and JSF, in both servlet and portlet environments. If you have a business process (or processes) that would benefit from a conversational model as opposed to a purely request model, then SWF may be the solution.

SWF allows you to capture logical page flows as self-contained modules that are reusable in different situations, and as such is ideal for building web application modules that guide the user through controlled navigations that drive business processes.

For more information about SWF, consult the Spring WebFlow site.

The dual phases of portlet requests are one of the real strengths of the JSR-168 specification. For example, dynamic search results can be updated routinely on the display without the user explicitly rerunning the search. Most other portlet MVC frameworks attempt to completely hide the two phases from the developer and make it look as much like traditional servlet development as possible - we think this approach removes one of the main benefits of using portlets. So, the separation of the two phases is preserved throughout the Spring Portlet MVC framework. The primary manifestation of this approach is that where the servlet version of the MVC classes

will have one method that deals with the request, the portlet version of the MVC classes will have two methods that deal with the request: one for the action phase and one for the render phase. For example, where the servlet version of `AbstractController` has the `handleRequestInternal(..)` method, the portlet version of `AbstractController` has `handleActionRequestInternal(..)` and `handleRenderRequestInternal(..)` methods.

The framework is designed around a `DispatcherPortlet` that dispatches requests to handlers, with configurable handler mappings and view resolution, just as the `DispatcherServlet` in the web framework does. File upload is also supported in the same way.

Locale resolution and theme resolution are not supported in Portlet MVC - these areas are in the purview of the portal/portlet-container and are not appropriate at the Spring level. However, all mechanisms in Spring that depend on the locale (such as internationalization of messages) will still function properly because `DispatcherPortlet` exposes the current locale in the same way as `DispatcherServlet`.

16.1.1. Controllers - The C in MVC

The default handler is still a very simple `Controller` interface, offering just two methods:

- `void handleActionRequest(request,response)`
- `ModelAndView handleRenderRequest(request,response)`

The framework also includes most of the same controller implementation hierarchy, such as `AbstractController`, `SimpleFormController`, and so on. Data binding, command object usage, model handling, and view resolution are all the same as in the servlet framework.

16.1.2. Views - The V in MVC

All the view rendering capabilities of the servlet framework are used directly via a special bridge servlet named `ViewRendererServlet`. By using this servlet, the portlet request is converted into a servlet request and the view can be rendered using the entire normal servlet infrastructure. This means all the existing renderers, such as JSP, Velocity, etc., can still be used within the portlet.

16.1.3. Web-scoped beans

Spring Portlet MVC supports beans whose lifecycle is scoped to the current HTTP request or HTTP `Session` (both normal and global). This is not a specific feature of Spring Portlet MVC itself, but rather of the `WebApplicationContext` container(s) that Spring Portlet MVC uses. These bean scopes are described in detail in the section entitled Section 3.4.4, “The other scopes”

Note

The Spring distribution ships with a complete Spring Portlet MVC sample application that demonstrates all of the features and functionality of the Spring Portlet MVC framework. This 'petportal' application can be found in the `'samples/petportal'` directory of the full Spring distribution.

16.2. The `DispatcherPortlet`

Portlet MVC is a request-driven web MVC framework, designed around a portlet that dispatches requests to controllers and offers other functionality facilitating the development of portlet applications. Spring's `DispatcherPortlet` however, does more than just that. It is completely integrated with the Spring `ApplicationContext` and allows you to use every other feature Spring has.

Like ordinary portlets, the `DispatcherPortlet` is declared in the `portlet.xml` of your web application:

```
<portlet>
        <portlet-name>sample</portlet-name>
        <portlet-class>org.springframework.web.portlet.DispatcherPortlet</portlet-class>
        <supports>
                <mime-type>text/html</mime-type>
                <portlet-mode>view</portlet-mode>
        </supports>
        <portlet-info>
                <title>Sample Portlet</title>
        </portlet-info>
</portlet>
```

The `DispatcherPortlet` now needs to be configured.

In the Portlet MVC framework, each `DispatcherPortlet` has its own `WebApplicationContext`, which inherits all the beans already defined in the Root `WebApplicationContext`. These inherited beans can be overridden in the portlet-specific scope, and new scope- specific beans can be defined local to a given portlet instance.

The framework will, on initialization of a `DispatcherPortlet`, look for a file named `[portlet-name]-portlet.xml` in the `WEB-INF` directory of your web application and create the beans defined there (overriding the definitions of any beans defined with the same name in the global scope).

The config location used by the `DispatcherPortlet` can be modified through a portlet initialization parameter (see below for details).

The Spring `DispatcherPortlet` has a few special beans it uses, in order to be able to process requests and render the appropriate views. These beans are included in the Spring framework and can be configured in the `WebApplicationContext`, just as any other bean would 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 `DispatcherPortlet`. For most of the beans, defaults are provided so you don't have to worry about configuring them.

Table 16.1. Special beans in the `WebApplicationContext`

Expression	Explanation
handler mapping(s)	(Section 16.5, “Handler mappings”) a list of pre- and post-processors and controllers that will be executed if they match certain criteria (for instance a matching portlet mode specified with the controller)
controller(s)	(Section 16.4, “Controllers”) the beans providing the actual functionality (or at least, access to the functionality) as part of the MVC triad
view resolver	(Section 16.6, “Views and resolving them”) capable of resolving view names to view definitions
multipart resolver	(Section 16.7, “Multipart (file upload) support”) offers functionality to process file uploads from HTML forms
handler exception resolver	(Section 16.8, “Handling exceptions”) offers functionality to map exceptions to views or implement other more complex exception handling code

When a `DispatcherPortlet` is setup for use and a request comes in for that specific `DispatcherPortlet`, it starts processing the request. The list below describes the complete process a request goes through if handled by a `DispatcherPortlet`:

1. The locale returned by `PortletRequest.getLocale()` is bound to the request to let elements in the process resolve the locale to use when processing the request (rendering the view, preparing data, etc.).

2. If a multipart resolver is specified and this is an `ActionRequest`, the request is inspected for multiparts and if they are found, it is wrapped in a `MultipartActionRequest` for further processing by other elements in the process. (See Section 16.7, “Multipart (file upload) support” for further information about multipart handling).

3. An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (pre- processors, post-processors, controllers) will be executed in order to prepare a model.

4. If a model is returned, the view is rendered, using the view resolver that has been configured with the `WebApplicationContext`. If no model is returned (which could be due to a pre- or post-processor intercepting the request, for example, for security reasons), no view is rendered, since the request could already have been fulfilled.

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

You can customize Spring's `DispatcherPortlet` by adding context parameters in the `portlet.xml` file or portlet init-parameters. The possibilities are listed below.

Table 16.2. `DispatcherPortlet` initialization parameters

Parameter	Explanation
`contextClass`	Class that implements `WebApplicationContext`, which will be used to instantiate the context used by this portlet. If this parameter isn't specified, the `XmlPortletApplicationContext` 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 `[portlet-name]-portlet`.
`viewRendererUrl`	The URL at which `DispatcherPortlet` can access an instance of `ViewRendererServlet` (see Section 16.3, “The `ViewRendererServlet`”).

16.3. The `ViewRendererServlet`

The rendering process in Portlet MVC is a bit more complex than in Web MVC. In order to reuse all the view technologies from Spring Web MVC), we must convert the `PortletRequest` / `PortletResponse` to `HttpServletRequest` / `HttpServletResponse` and then call the `render` method of the `View`. To do this,

`DispatcherPortlet` uses a special servlet that exists for just this purpose: the `ViewRendererServlet`.

In order for `DispatcherPortlet` rendering to work, you must declare an instance of the `ViewRendererServlet` in the `web.xml` file for your web application as follows:

```
<servlet>
    <servlet-name>ViewRendererServlet</servlet-name>
    <servlet-class>org.springframework.web.servlet.ViewRendererServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>ViewRendererServlet</servlet-name>
    <url-pattern>/WEB-INF/servlet/view</url-pattern>
</servlet-mapping>
```

To perform the actual rendering, `DispatcherPortlet` does the following:

1. Binds the `WebApplicationContext` to the request as an attribute under the same `WEB_APPLICATION_CONTEXT_ATTRIBUTE` key that `DispatcherServlet` uses.
2. Binds the `Model` and `View` objects to the request to make them available to the `ViewRendererServlet`.
3. Constructs a `PortletRequestDispatcher` and performs an `include` using the `/WEB- INF/servlet/view` URL that is mapped to the `ViewRendererServlet`.

The `ViewRendererServlet` is then able to call the `render` method on the `View` with the appropriate arguments.

The actual URL for the `ViewRendererServlet` can be changed using `DispatcherPortlet`'s `viewRendererUrl` configuration parameter.

16.4. Controllers

The controllers in Portlet MVC are very similar to the Web MVC Controllers and porting code from one to the other should be simple.

The basis for the Portlet MVC controller architecture is the `org.springframework.web.portlet.mvc.Controller` interface, which is listed below.

```
public interface Controller {

    /**
     * Process the render request and return a ModelAndView object which the
     * DispatcherPortlet will render.
     */
    ModelAndView handleRenderRequest(RenderRequest request, RenderResponse response)
        throws Exception;

    /**
     * Process the action request. There is nothing to return.
     */
    void handleActionRequest(ActionRequest request, ActionResponse response)
        throws Exception;

}
```

As you can see, the Portlet `Controller` interface requires two methods that handle the two phases of a portlet request: the action request and the render request. The action phase should be capable of handling an action request and the render phase should be capable of handling a render request and returning an appropriate model and view. While the `Controller` interface is quite abstract, Spring Portlet MVC offers a lot of controllers that already contain a lot of the functionality you might need – most of these are very similar to controllers from

Spring Web MVC. The `Controller` interface just defines the most common functionality required of every controller - handling an action request, handling a render request, and returning a model and a view.

16.4.1. `AbstractController` and `PortletContentGenerator`

Of course, just a `Controller` interface isn't enough. To provide a basic infrastructure, all of Spring Portlet MVC's `Controllers` inherit from `AbstractController`, a class offering access to Spring's `ApplicationContext` and control over caching.

Table 16.3. Features offered by the `AbstractController`

Parameter	Explanation
`requireSession`	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 `SessionRequiredException`.
`synchronizeSession`	Use this if you want handling by this controller to be synchronized on the user's session. To be more specific, the extending controller will override the `handleRenderRequestInternal(..)` and `handleActionRequestInternal(..)` methods, which will be synchronized on the user's session if you specify this variable.
`renderWhenMinimized`	If you want your controller to actually render the view when the portlet is in a minimized state, set this to true. By default, this is set to false so that portlets that are in a minimized state don't display any content.
`cacheSeconds`	When you want a controller to override the default cache expiration defined for the portlet, specify a positive integer here. By default it is set to `-1`, which does not change the default caching. Setting it to `0` will ensure the result is never cached.

The `requireSession` and `cacheSeconds` properties are declared on the `PortletContentGenerator` class, which is the superclass of `AbstractController`) but are included here for completeness.

When using the `AbstractController` as a baseclass for your controllers (which is not recommended since there are a lot of other controllers that might already do the job for you) you only have to override either the `handleActionRequestInternal(ActionRequest, ActionResponse)` method or the `handleRenderRequestInternal(RenderRequest, RenderResponse)` method (or both), implement your logic, and return a `ModelAndView` object (in the case of `handleRenderRequestInternal`).

The default implementations of both `handleActionRequestInternal(..)` and `handleRenderRequestInternal(..)` throw a `PortletException`. This is consistent with the behavior of `GenericPortlet` from the JSR- 168 Specification API. So you only need to override the method that your controller is intended to handle.

Here is short example consisting of a class and a declaration in the web application context.

```
package samples;

import javax.portlet.RenderRequest;
import javax.portlet.RenderResponse;

import org.springframework.web.portlet.mvc.AbstractController;
import org.springframework.web.portlet.ModelAndView;
```

```
public class SampleController extends AbstractController {

    public ModelAndView handleRenderRequestInternal(
        RenderRequest request,
        RenderResponse response) throws Exception {

        ModelAndView mav = new ModelAndView("foo");
        mav.addObject("message", "Hello World!");
        return mav;
    }
}

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

The class above and the declaration in the web application context is all you need besides setting up a handler mapping (see Section 16.5, “Handler mappings”) to get this very simple controller working.

16.4.2. Other simple controllers

Although you can extend `AbstractController`, Spring Portlet MVC provides a number of concrete implementations which offer functionality that is commonly used in simple MVC applications.

The `ParameterizableViewController` is basically the same as the example above, except for the fact that you can specify the view name that it will return in the web application context (no need to hard-code the view name).

The `PortletModeNameViewController` uses the current mode of the portlet as the view name. So, if your portlet is in View mode (i.e. `PortletMode.VIEW`) then it uses "view" as the view name.

16.4.3. Command Controllers

Spring Portlet MVC has the exact same hierarchy of *command controllers* as Spring Web MVC. They provide a way to interact with data objects and dynamically bind parameters from the `PortletRequest` to the data object specified. Your data objects don't have to implement a framework-specific interface, so you can directly manipulate your persistent objects if you desire. Let's examine what command controllers are available, to get an overview 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 specify. This class does not offer form functionality, it does however offer validation features and lets you specify in the controller itself what to do with the command object 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 command object you retrieve in the controller. After a user has filled the form, `AbstractFormController` binds the fields, validates, and hands the object back to the controller to take appropriate action. Supported features are: invalid form submission (resubmission), validation, and normal form workflow. You implement methods to determine which views are used for form presentation and success. Use this controller if you need forms, but don't want to specify what views you're going to show the user in the application context.

- `SimpleFormController` - a concrete `AbstractFormController` that provides even more support when creating a form with a corresponding command object. The `SimpleFormController` lets you specify a command object, a viewname for the form, a viewname for the page you want to show the user when form

submission has succeeded, and more.

- `AbstractWizardFormController` – a concrete `AbstractFormController` that provides a wizard-style interface for editing the contents of a command object across multiple display pages. Supports multiple user actions: finish, cancel, or page change, all of which are easily specified in request parameters from the view.

These command controllers are quite powerful, but they do require a detailed understanding of how they operate in order to use them efficiently. Carefully review the Javadocs for this entire hierarchy and then look at some sample implementations before you start using them.

16.4.4. `PortletWrappingController`

Instead of developing new controllers, it is possible to use existing portlets and map requests to them from a `DispatcherPortlet`. Using the `PortletWrappingController`, you can instantiate an existing `Portlet` as a `Controller` as follows:

```
<bean id="wrappingController"
      class="org.springframework.web.portlet.mvc.PortletWrappingController">
    <property name="portletClass" value="sample.MyPortlet"/>
    <property name="portletName" value="my-portlet"/>
    <property name="initParameters">
        <value>
            config=/WEB-INF/my-portlet-config.xml
        </value>
    </property>
</bean>
```

This can be very valuable since you can then use interceptors to pre-process and post-process requests going to these portlets. Since JSR-168 does not support any kind of filter mechanism, this is quite handy. For example, this can be used to wrap the Hibernate `OpenSessionInViewInterceptor` around a MyFaces JSF Portlet.

16.5. Handler mappings

Using a handler mapping you can map incoming portlet requests to appropriate handlers. There are some handler mappings you can use out of the box, for example, the `PortletModeHandlerMapping`, but let's first examine the general concept of a `HandlerMapping`.

Note: We are intentionally using the term “Handler” here instead of “Controller”. `DispatcherPortlet` is designed to be used with other ways to process requests than just Spring Portlet MVC’s own Controllers. A Handler is any Object that can handle portlet requests. Controllers are an example of Handlers, and they are of course the default. To use some other framework with `DispatcherPortlet`, a corresponding implementation of `HandlerAdapter` is all that is needed.

The functionality a basic `HandlerMapping` provides is the delivering of a `HandlerExecutionChain`, which must contain the handler that matches the incoming request, and may also contain a list of handler interceptors that are applied to the request. When a request comes in, the `DispatcherPortlet` will hand it over to the handler mapping to let it inspect the request and come up with an appropriate `HandlerExecutionChain`. Then the `DispatcherPortlet` will execute the handler and interceptors in the chain (if any). These concepts are all exactly the same as in Spring Web MVC.

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 into a custom `HandlerMapping`. Think of a custom handler mapping that chooses a handler not only based on the portlet mode of the request coming in, but also on a specific state of the session associated with the request.

In Spring Web MVC, handler mappings are commonly based on URLs. Since there is really no such thing as a URL within a Portlet, we must use other mechanisms to control mappings. The two most common are the portlet mode and a request parameter, but anything available to the portlet request can be used in a custom handler mapping.

The rest of this section describes three of Spring Portlet MVC's most commonly used handler mappings. They all extend `AbstractHandlerMapping` and share the following properties:

- `interceptors`: The list of interceptors to use. `HandlerInterceptors` are discussed in Section 16.5.4, “Adding `HandlerInterceptors`”.
- `defaultHandler`: The default handler to use, when this handler mapping does not result in a matching handler.
- `order`: Based on the value of the order property (see the `org.springframework.core.Ordered` interface), Spring will sort all handler mappings available in the context and apply the first matching handler.
- `lazyInitHandlers`: Allows for lazy initialization of singleton handlers (prototype handlers are always lazily initialized). Default value is false. This property is directly implemented in the three concrete Handlers.

16.5.1. `PortletModeHandlerMapping`

This is a simple handler mapping that maps incoming requests based on the current mode of the portlet (e.g. ‘view’, ‘edit’, ‘help’). An example:

```
<bean id="portletModeHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="viewHandler"/>
            <entry key="edit" value-ref="editHandler"/>
            <entry key="help" value-ref="helpHandler"/>
        </map>
    </property>
</bean>
```

16.5.2. `ParameterHandlerMapping`

If we need to navigate around to multiple controllers without changing portlet mode, the simplest way to do this is with a request parameter that is used as the key to control the mapping.

`ParameterHandlerMapping` uses the value of a specific request parameter to control the mapping. The default name of the parameter is `'action'`, but can be changed using the `'parameterName'` property.

The bean configuration for this mapping will look something like this:

```
<bean id="parameterHandlerMapping"
      class="org.springframework.web.portlet.handler.ParameterHandlerMapping”>
    <property name="parameterMap">
        <map>
            <entry key="add" value-ref="addItemHandler"/>
            <entry key="edit" value-ref="editItemHandler"/>
            <entry key="delete" value-ref="deleteItemHandler"/>
        </map>
    </property>
</bean>
```

16.5.3. `PortletModeParameterHandlerMapping`

The most powerful built-in handler mapping, `PortletModeParameterHandlerMapping` combines the capabilities of the two previous ones to allow different navigation within each portlet mode.

Again the default name of the parameter is "action", but can be changed using the `parameterName` property.

By default, the same parameter value may not be used in two different portlet modes. This is so that if the portal itself changes the portlet mode, the request will no longer be valid in the mapping. This behavior can be changed by setting the `allowDupParameters` property to true. However, this is not recommended.

The bean configuration for this mapping will look something like this:

```
<bean id="portletModeParameterHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeParameterHandlerMapping">
    <property name="portletModeParameterMap">
        <map>
            <entry key="view"> <!-- 'view' portlet mode -->
                <map>
                    <entry key="add" value-ref="addItemHandler"/>
                    <entry key="edit" value-ref="editItemHandler"/>
                    <entry key="delete" value-ref="deleteItemHandler"/>
                </map>
            </entry>
            <entry key="edit"> <!-- 'edit' portlet mode -->
                <map>
                    <entry key="prefs" value-ref="prefsHandler"/>
                    <entry key="resetPrefs" value-ref="resetPrefsHandler"/>
                </map>
            </entry>
        </map>
    </property>
</bean>
```

This mapping can be chained ahead of a `PortletModeHandlerMapping`, which can then provide defaults for each mode and an overall default as well.

16.5.4. Adding `HandlerInterceptorS`

Spring's handler mapping mechanism has a notion of handler interceptors, which can be extremely useful when you want to apply specific functionality to certain requests, for example, checking for a principal. Again Spring Portlet MVC implements these concepts in the same way as Web MVC.

Interceptors located in the handler mapping must implement `HandlerInterceptor` from the `org.springframework.web.portlet` package. Just like the servlet version, this interface defines three methods: one that will be called before the actual handler will be executed (`preHandle`), one that will be called after the handler is executed (`postHandle`), and one that is called after the complete request has finished (`afterCompletion`). These three methods should provide enough flexibility to do all kinds of pre- and post-processing.

The `preHandle` method returns a boolean value. You can use this method to break or continue the processing of the execution chain. When this method returns `true`, the handler execution chain will continue. When it returns `false`, the `DispatcherPortlet` assumes the interceptor itself has taken care of requests (and, for example, rendered an appropriate view) and does not continue executing the other interceptors and the actual handler in the execution chain.

The `postHandle` method is only called on a `RenderRequest`. The `preHandle` and `afterCompletion` methods are called on both an `ActionRequest` and a `RenderRequest`. If you need to execute logic in these methods for just one type of request, be sure to check what kind of request it is before processing it.

16.5.5. `HandlerInterceptorAdapter`

As with the servlet package, the portlet package has a concrete implementation of `HandlerInterceptor` called `HandlerInterceptorAdapter`. This class has empty versions of all the methods so that you can inherit from this class and implement just one or two methods when that is all you need.

16.5.6. `ParameterMappingInterceptor`

The portlet package also has a concrete interceptor named `ParameterMappingInterceptor` that is meant to be used directly with `ParameterHandlerMapping` and `PortletModeParameterHandlerMapping`. This interceptor will cause the parameter that is being used to control the mapping to be forwarded from an `ActionRequest` to the subsequent `RenderRequest`. This will help ensure that the `RenderRequest` is mapped to the same Handler as the `ActionRequest`. This is done in the `preHandle` method of the interceptor, so you can still modify the parameter value in your handler to change where the `RenderRequest` will be mapped.

Be aware that this interceptor is calling `setRenderParameter` on the `ActionResponse`, which means that you cannot call `sendRedirect` in your handler when using this interceptor. If you need to do external redirects then you will either need to forward the mapping parameter manually or write a different interceptor to handle this for you.

16.6. Views and resolving them

As mentioned previously, Spring Portlet MVC directly reuses all the view technologies from Spring Web MVC. This includes not only the various `View` implementations themselves, but also the `ViewResolver` implementations. For more information, refer to the sections entitled Chapter 14, *Integrating view technologies* and Section 13.5, “Views and resolving them” respectively.

A few items on using the existing `View` and `ViewResolver` implementations are worth mentioning:

- Most portals expect the result of rendering a portlet to be an HTML fragment. So, things like JSP/JSTL, Velocity, FreeMarker, and XSLT all make sense. But it is unlikely that views that return other document types will make any sense in a portlet context.
- There is no such thing as an HTTP redirect from within a portlet (the `sendRedirect(..)` method of `ActionResponse` cannot be used to stay within the portal). So, `RedirectView` and use of the `'redirect:'` prefix will **not** work correctly from within Portlet MVC.
- It may be possible to use the `'forward:'` prefix from within Portlet MVC. However, remember that since you are in a portlet, you have no idea what the current URL looks like. This means you cannot use a relative URL to access other resources in your web application and that you will have to use an absolute URL.

Also, for JSP development, the new Spring Taglib and the new Spring Form Taglib both work in portlet views in exactly the same way that they work in servlet views.

16.7. Multipart (file upload) support

Spring Portlet MVC has built-in multipart support to handle file uploads in portlet applications, just like Web MVC does. The design for the multipart support is done with pluggable `PortletMultipartResolver` objects, defined in the `org.springframework.web.portlet.multipart` package. Spring provides a `PortletMultipartResolver` for use with Commons FileUpload. How uploading files is supported will be

described in the rest of this section.

By default, no multipart handling will be done by Spring Portlet MVC, as some developers will want to handle multiparts themselves. You will have to enable it yourself by adding a multipart resolver to the web application's context. After you have done that, `DispatcherPortlet` will inspect each request to see if it contains a multipart. If no multipart is found, the request will continue as expected. However, if a multipart is found in the request, the `PortletMultipartResolver` that has been declared in your context will be used. After that, the multipart attribute in your request will be treated like any other attribute.

Note

Any configured `PortletMultipartResolver` bean *must* have the following id (or name): "`portletMultipartResolver`". If you have defined your `PortletMultipartResolver` with any other name, then the `DispatcherPortlet` will *not* find your `PortletMultipartResolver`, and consequently no multipart support will be in effect.

16.7.1. Using the `PortletMultipartResolver`

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

```
<bean id="portletMultipartResolver"
    class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver">

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

Of course you also need to put the appropriate jars in your classpath for the multipart resolver to work. In the case of the `CommonsMultipartResolver`, you need to use `commons-fileupload.jar`. Be sure to use at least version 1.1 of Commons FileUpload as previous versions do not support JSR-168 Portlet applications.

Now that you have seen how to set Portlet MVC up to handle multipart requests, let's talk about how to actually use it. When `DispatcherPortlet` detects a multipart request, it activates the resolver that has been declared in your context and hands over the request. What the resolver then does is wrap the current `ActionRequest` into a `MultipartActionRequest` that has support for multipart file uploads. Using the `MultipartActionRequest` you can get information about the multiparts contained by this request and actually get access to the multipart files themselves in your controllers.

Note that you can only receive multipart file uploads as part of an `ActionRequest`, not as part of a `RenderRequest`.

16.7.2. Handling a file upload in a form

After the `PortletMultipartResolver` has finished doing its job, the request will be processed like any other. To use it, you create a form with an upload field (see immediately below), then let Spring bind the file onto your form (backing object). To actually let the user upload a file, we have to create a (JSP/HTML) form:

```
<h1>Please upload a file</h1>
<form method="post" action="<portlet:actionURL/>" enctype="multipart/form-data">
    <input type="file" name="file"/>
    <input type="submit"/>
</form>
```

As you can see, we've created a field named “file” after the property of the bean that holds the `byte[]`.

Furthermore we've added the encoding attribute (`enctype="multipart/form-data"`), which is necessary to let the browser know how to encode the multipart fields (do not forget this!).

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 objects you have to register a custom editor with the `PortletRequestDataBinder`. There are a couple of editors available for handling files and setting the results on an object. There's a `StringMultipartFileEditor` capable of converting files to Strings (using a user-defined character set) and there is a `ByteArrayMultipartFileEditor` which converts files to byte arrays. They function just as the `CustomDateEditor` does.

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

```
<bean id="portletMultipartResolver"
      class="org.springframework.web.portlet.multipart.CommonsPortletMultipartResolver"/>

<bean id="portletModeHandlerMapping"
      class="org.springframework.web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
        <map>
            <entry key="view" value-ref="fileUploadController"/>
        </map>
    </property>
</bean>

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

After that, create the controller and the actual class to hold the file property.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors)
        throws Exception {

        // 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
        }

        // do something with the file here
    }

    protected void initBinder(
            PortletRequest request, PortletRequestDataBinder binder)
        throws Exception {
        // to actually be able to convert Multipart instance to byte[]
        // we have to register a custom editor
        binder.registerCustomEditor(byte[].class, new ByteArrayMultipartFileEditor());
        // now Spring knows how to handle multipart object and convert
    }
}

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 registers a custom editor to let Spring know how to actually convert the multipart objects the resolver has found to properties specified by the bean. In this example, nothing is done with the `byte[]` property of the bean itself, but in practice you can do whatever you want (save it in a database, mail it to somebody, etc).

An equivalent example in which a file is bound straight to a String-typed property on a (form backing) object might look like this:

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
        BindException errors) throws Exception {

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

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

        // do something with the file here
    }

    protected void initBinder(
        PortletRequest request, PortletRequestDataBinder binder) throws Exception {

        // to actually be able to convert Multipart instance to a String
        // we have to register a custom editor
        binder.registerCustomEditor(String.class,
            new StringMultipartFileEditor());
        // now Spring knows how to handle multipart objects and convert
    }
}

public class FileUploadBean {

    private String file;

    public void setFile(String file) {
        this.file = file;
    }

    public String getFile() {
        return file;
    }
}
```

Of course, this last example only makes (logical) sense in the context of uploading a plain text file (it wouldn't work so well in the case of uploading an image file).

The third (and final) option is where one binds directly to a `MultipartFile` property declared on the (form backing) object's class. In this case one does not need to register any custom property editor because there is no type conversion to be performed.

```
public class FileUploadController extends SimpleFormController {

    public void onSubmitAction(
        ActionRequest request,
        ActionResponse response,
        Object command,
```

```
        BindException errors) throws Exception {

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

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

        // do something with the file here
    }
}

public class FileUploadBean {

    private MultipartFile file;

    public void setFile(MultipartFile file) {
        this.file = file;
    }

    public MultipartFile getFile() {
        return file;
    }
}
```

16.8. Handling exceptions

Just like Web MVC, Portlet MVC provides `HandlerExceptionResolvers` to ease the pain of unexpected exceptions occurring while your request is being processed by a handler that matched the request. Portlet MVC also provides the same concrete `SimpleMappingExceptionResolver` that enables you to take the class name of any exception that might be thrown and map it to a view name.

16.9. Portlet application deployment

The process of deploying a Spring Portlet MVC application is no different than deploying any JSR-168 Portlet application. However, this area is confusing enough in general that it is worth talking about here briefly.

Generally, the portal/portlet-container runs in one webapp in your servlet-container and your portlets run in another webapp in your servlet-container. In order for the portlet-container webapp to make calls into your portlet webapp it must make cross-context calls to a well-known servlet that provides access to the portlet services defined in your `portlet.xml` file.

The JSR-168 specification does not specify exactly how this should happen, so each portlet-container has its own mechanism for this, which usually involves some kind of “deployment process” that makes changes to the portlet webapp itself and then registers the portlets within the portlet-container.

At a minimum, the `web.xml` file in your portlet webapp is modified to inject the well-known servlet that the portlet- container will call. In some cases a single servlet will service all portlets in the webapp, in other cases there will be an instance of the servlet for each portlet.

Some portlet-containers will also inject libraries and/or configuration files into the webapp as well. The portlet-container must also make its implementation of the Portlet JSP Tag Library available to your webapp.

The bottom line is that it is important to understand the deployment needs of your target portal and make sure they are met (usually by following the automated deployment process it provides). Be sure to carefully review the documentation from your portal for this process.

Once you have deployed your portlet, review the resulting `web.xml` file for sanity. Some older portals have been known to corrupt the definition of the `ViewRendererServlet`, thus breaking the rendering of your portlets.

Part IV. Integration

This part of the reference documentation covers the Spring Framework's integration with a number of J2EE (and related) technologies.

- Chapter 17, *Remoting and web services using Spring*
- Chapter 18, *Enterprise Java Bean (EJB) integration*
- Chapter 19, *JMS*
- Chapter 20, *JMX*
- Chapter 21, *JCA CCI*
- Chapter 22, *Email*
- Chapter 23, *Scheduling and Thread Pooling*
- Chapter 24, *Dynamic language support*
- Chapter 25, *Annotations and Source Level Metadata Support*

Chapter 17. Remoting and web services using Spring

17.1. Introduction

Spring features integration classes for remoting support using various technologies. The remoting support eases the development of remote-enabled services, implemented by your usual (Spring) POJOs. Currently, Spring supports four remoting technologies:

- *Remote Method Invocation (RMI)*. Through the use of the `RmiProxyFactoryBean` and the `RmiServiceExporter` Spring supports both traditional RMI (with `java.rmi.Remote` interfaces and `java.rmi.RemoteException`) and transparent remoting via RMI invokers (with any Java interface).
- *Spring's HTTP invoker*. Spring provides a special remoting strategy which allows for Java serialization via HTTP, supporting any Java interface (just like the RMI invoker). The corresponding support classes are `HttpInvokerProxyFactoryBean` and `HttpInvokerServiceExporter`.
- *Hessian*. By using the `HessianProxyFactoryBean` and the `HessianServiceExporter` you can transparently expose your services using the lightweight binary HTTP-based protocol provided by Caucho.
- *Burlap*. Burlap is Caucho's XML-based alternative for Hessian. Spring provides support classes such as `BurlapProxyFactoryBean` and `BurlapServiceExporter`.
- *JAX RPC*. Spring provides remoting support for web services via JAX-RPC.
- *JMS*. Remoting using JMS as the underlying protocol is supported via the `JmsInvokerServiceExporter` and `JmsInvokerProxyFactoryBean` classes.

While discussing the remoting capabilities of Spring, we'll use the following domain model and corresponding services:

```
public class Account implements Serializable{
  private String name;

  public String getName();

  public void setName(String name) {
    this.name = name;
  }
}
```

```
public interface AccountService {

  public void insertAccount(Account account);

  public List getAccounts(String name);
}
```

```
public interface RemoteAccountService extends Remote {

  public void insertAccount(Account account) throws RemoteException;

  public List getAccounts(String name) throws RemoteException;
}
```

```
// the implementation doing nothing at the moment
public class AccountServiceImpl implements AccountService {

  public void insertAccount(Account acc) {
    // do something...
  }

  public List getAccounts(String name) {
    // do something...
  }
}
```

We will start exposing the service to a remote client by using RMI and talk a bit about the drawbacks of using RMI. We'll then continue to show an example using Hessian as the protocol.

17.2. Exposing services using RMI

Using Spring's support for RMI, you can transparently expose your services through the RMI infrastructure. After having this set up, you basically have a configuration similar to remote EJBs, except for the fact that there is no standard support for security context propagation or remote transaction propagation. Spring does provide hooks for such additional invocation context when using the RMI invoker, so you can for example plug in security frameworks or custom security credentials here.

17.2.1. Exporting the service using the `RmiServiceExporter`

Using the `RmiServiceExporter`, we can expose the interface of our AccountService object as RMI object. The interface can be accessed by using `RmiProxyFactoryBean`, or via plain RMI in case of a traditional RMI service. The `RmiServiceExporter` explicitly supports the exposing of any non-RMI services via RMI invokers.

Of course, we first have to set up our service in the Spring container:

```
<bean id="accountService" class="example.AccountServiceImpl">
    <!-- any additional properties, maybe a DAO? -->
</bean>
```

Next we'll have to expose our service using the `RmiServiceExporter`:

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
        <!-- does not necessarily have to be the same name as the bean to be exported -->
        <property name="serviceName" value="AccountService"/>
        <property name="service" ref="accountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
        <!-- defaults to 1099 -->
        <property name="registryPort" value="1199"/>
</bean>
```

As you can see, we're overriding the port for the RMI registry. Often, your application server also maintains an RMI registry and it is wise to not interfere with that one. Furthermore, the service name is used to bind the service under. So right now, the service will be bound at `'rmi://HOST:1199/AccountService'`. We'll use the URL later on to link in the service at the client side.

Note

The `servicePort` property has been omitted (it defaults to 0). This means that an anonymous port will be used to communicate with the service.

17.2.2. Linking in the service at the client

Our client is a simple object using the `AccountService` to manage accounts:

```
public class SimpleObject {

  private AccountService accountService;

  public void setAccountService(AccountService accountService) {
    this.accountService = accountService;
  }
}
```

To link in the service on the client, we'll create a separate Spring container, containing the simple object and the service linking configuration bits:

```
<bean class="example.SimpleObject">
        <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
        <property name="serviceUrl" value="rmi://HOST:1199/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

That's all we need to do to support the remote account service on the client. Spring will transparently create an invoker and remotely enable the account service through the `RmiServiceExporter`. At the client we're linking it in using the `RmiProxyFactoryBean`.

17.3. Using Hessian or Burlap to remotely call services via HTTP

Hessian offers a binary HTTP-based remoting protocol. It is developed by Caucho and more information about Hessian itself can be found at http://www.caucho.com.

17.3.1. Wiring up the `DispatcherServlet` for Hessian

Hessian communicates via HTTP and does so using a custom servlet. Using Spring's `DispatcherServlet` principles, you can easily wire up such a servlet exposing your services. First we'll have to create a new servlet in your application (this an excerpt from `'web.xml'`):

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

<servlet-mapping>
        <servlet-name>remoting</servlet-name>
        <url-pattern>/remoting/*</url-pattern>
</servlet-mapping>
```

You're probably familiar with Spring's `DispatcherServlet` principles and if so, you know that now you'll have to create a Spring container configuration resource named `'remoting-servlet.xml'` (after the name of your servlet) in the `'WEB-INF'` directory. The application context will be used in the next section.

17.3.2. Exposing your beans by using the `HessianServiceExporter`

In the newly created application context called `remoting-servlet.xml`, we'll create a `HessianServiceExporter` exporting your services:

```
<bean id="accountService" class="example.AccountServiceImpl">
  <!-- any additional properties, maybe a DAO? -->
</bean>

<bean name="/AccountService" class="org.springframework.remoting.caucho.HessianServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

Now we're ready to link in the service at the client. No explicit handler mapping is specified, mapping request URLs onto services, so `BeanNameUrlHandlerMapping` will be used: hence, the service will be exported at the URL indicated through its bean name: `'http://HOST:8080/remoting/AccountService'`.

17.3.3. Linking in the service on the client

Using the we can link in the service at the client. The same principles apply as with the RMI example. We'll create a separate bean factory or application context and mention the following beans where the `SimpleObject` is using the `AccountService` to manage accounts:

```

<bean class="example.SimpleObject">
  <property name="accountService" ref="accountService"/>
</bean>

<bean id="accountService" class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
        <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

17.3.4. Using Burlap

We won't discuss Burlap, the XML-based equivalent of Hessian, in detail here, since it is configured and set up in exactly the same way as the Hessian variant explained above. Just replace the word `Hessian` with `Burlap` and you're all set to go.

17.3.5. Applying HTTP basic authentication to a service exposed through Hessian or Burlap

One of the advantages of Hessian and Burlap is that we can easily apply HTTP basic authentication, because both protocols are HTTP-based. Your normal HTTP server security mechanism can easily be applied through using the `web.xml` security features, for example. Usually, you don't use per-user security credentials here, but rather shared credentials defined at the `Hessian/BurlapProxyFactoryBean` level (similar to a JDBC `DataSource`).

```
<bean class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping">
   <property name="interceptors">
      <list>
         <ref bean="authorizationInterceptor"/>
      </list>
   </property>
</bean>

<bean id="authorizationInterceptor"
```

```
   class="org.springframework.web.servlet.handler.UserRoleAuthorizationInterceptor">
   <property name="authorizedRoles">
      <list>
         <value>administrator</value>
         <value>operator</value>
      </list>
   </property>
</bean>
```

This an example where we explicitly mention the `BeanNameUrlHandlerMapping` and set an interceptor allowing only administrators and operators to call the beans mentioned in this application context.

Note

Of course, this example doesn't show a flexible kind of security infrastructure. For more options as far as security is concerned, have a look at the Acegi Security System for Spring, to be found at http://acegisecurity.sourceforge.net.

17.4. Exposing services using HTTP invokers

As opposed to Burlap and Hessian, which are both lightweight protocols using their own slim serialization mechanisms, Spring Http invokers use the standard Java serialization mechanism to expose services through HTTP. This has a huge advantage if your arguments and return types are complex types that cannot be serialized using the serialization mechanisms Hessian and Burlap use (refer to the next section for more considerations when choosing a remoting technology).

Under the hood, Spring uses either the standard facilities provided by J2SE to perform HTTP calls or Commons `HttpClient`. Use the latter if you need more advanced and easy-to-use functionality. Refer to jakarta.apache.org/commons/httpclient for more info.

17.4.1. Exposing the service object

Setting up the HTTP invoker infrastructure for a service objects much resembles the way you would do using Hessian or Burlap. Just as Hessian support provides the `HessianServiceExporter`, Spring's HttpInvoker support provides the `org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter`. To expose the `AccountService` (mentioned above), the following configuration needs to be in place:

```
<bean name="/AccountService" class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

17.4.2. Linking in the service at the client

Again, linking in the service from the client much resembles the way you would do it when using Hessian or Burlap. Using a proxy, Spring will be able to translate your calls to HTTP POST requests to the URL pointing to the exported service.

```
<bean id="httpInvokerProxy" class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl" value="http://remotehost:8080/remoting/AccountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

As mentioned before, you can choose what HTTP client you want to use. By default, the `HttpInvokerProxy` uses the J2SE HTTP functionality, but you can also use the Commons `HttpClient` by setting the `httpInvokerRequestExecutor` property:

```
<property name="httpInvokerRequestExecutor">
        <bean class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerRequestExecutor"/>
</property>
```

17.5. Web services

Spring has support for:

- Exposing services using JAX-RPC
- Accessing web services

In addition to stock support for JAX-RPC in Spring Core, the Spring portfolio also features Spring Web Services, a solution for contract-first, document-driven web services. Last but not least, XFire also allows you to export Spring-managed beans as a web service.

17.5.1. Exposing services using JAX-RPC

Spring has a convenience base class for JAX-RPC servlet endpoint implementations - `ServletEndpointSupport`. To expose our `AccountService` we extend Spring's `ServletEndpointSupport` class and implement our business logic here, usually delegating the call to the business layer.

```
/**
 * JAX-RPC compliant RemoteAccountService implementation that simply delegates
 * to the AccountService implementation in the root web application context.
 *
 * This wrapper class is necessary because JAX-RPC requires working with
 * RMI interfaces. If an existing service needs to be exported, a wrapper that
 * extends ServletEndpointSupport for simple application context access is
 * the simplest JAX-RPC compliant way.
 *
 * This is the class registered with the server-side JAX-RPC implementation.
 * In the case of Axis, this happens in "server-config.wsdd" respectively via
 * deployment calls. The Web Service tool manages the life-cycle of instances
 * of this class: A Spring application context can just be accessed here.
 */
public class AccountServiceEndpoint extends ServletEndpointSupport implements RemoteAccountService {

    private AccountService biz;

    protected void onInit() {
        this.biz = (AccountService) getWebApplicationContext().getBean("accountService");
    }

    public void insertAccount(Account acc) throws RemoteException {
        biz.insertAccount(acc);
    }

    public Account[] getAccounts(String name) throws RemoteException {
        return biz.getAccounts(name);
    }
}
```

Our `AccountServletEndpoint` needs to run in the same web application as the Spring context to allow for access to Spring's facilities. In case of Axis, copy the `AxisServlet` definition into your `'web.xml'`, and set up

the endpoint in `'server-config.wsdd'` (or use the deploy tool). See the sample application JPetStore where the `OrderService` is exposed as a Web Service using Axis.

17.5.2. Accessing web services

Spring has two factory beans to create web service proxies, namely `LocalJaxRpcServiceFactoryBean` and `JaxRpcPortProxyFactoryBean`. The former can only return a JAX-RPC service class for us to work with. The latter is the full fledged version that can return a proxy that implements our business service interface. In this example we use the latter to create a proxy for the `AccountService` endpoint we exposed in the previous paragraph. You will see that Spring has great support for web services requiring little coding efforts - most of the setup is done in the Spring configuration file as usual:

```
<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
    <property name="serviceInterface" value="example.RemoteAccountService"/>
    <property name="wsdlDocumentUrl" value="http://localhost:8080/account/services/accountService?WSDL"/>
    <property name="namespaceUri" value="http://localhost:8080/account/services/accountService"/>
    <property name="serviceName" value="AccountService"/>
    <property name="portName" value="AccountPort"/>
</bean>
```

Where `serviceInterface` is our remote business interface the clients will use. `wsdlDocumentUrl` is the URL for the WSDL file. Spring needs this a startup time to create the JAX-RPC Service. `namespaceUri` corresponds to the targetNamespace in the .wsdl file. `serviceName` corresponds to the serivce name in the .wsdl file. `portName` corresponds to the port name in the .wsdl file.

Accessing the Web Service is now very easy as we have a bean factory for it that will expose it as `RemoteAccountService` interface. We can wire this up in Spring:

```
<bean id="client" class="example.AccountClientImpl">
    ...
    <property name="service" ref="accountWebService"/>
</bean>
```

From the client code we can access the web service just as if it was a normal class, except that it throws `RemoteException`.

```
public class AccountClientImpl {

    private RemoteAccountService service;

    public void setService(RemoteAccountService service) {
        this.service = service;
    }

    public void foo() {
       try {
           service.insertAccount(...);
        } catch (RemoteException ex) {
           // ouch
        }
     }
}
```

We can get rid of the checked `RemoteException` since Spring supports automatic conversion to its corresponding unchecked `RemoteException`. This requires that we provide a non-RMI interface also. Our configuration is now:

```
<bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
    <property name="serviceInterface" value="example.AccountService"/>
    <property name="portInterface" value="example.RemoteAccountService"/>
</bean>
```

Where `serviceInterface` is changed to our non RMI interface. Our RMI interface is now defined using the property `portInterface`. Our client code can now avoid handling `java.rmi.RemoteException`:

```
public class AccountClientImpl {

    private AccountService service;

    public void setService(AccountService service) {
        this.service = service;
    }

    public void foo() {
        service.insertAccount(...);
    }
}
```

17.5.3. Register Bean Mappings

To transfer complex objects over the wire such as `Account` we must register bean mappings on the client side.

Note

On the server side using Axis registering bean mappings is usually done in the `'server-config.wsdd'` file.

We will use Axis to register bean mappings on the client side. To do this we need to register the bean mappings programmatically:

```
public class AxisPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {

   protected void postProcessJaxRpcService(Service service) {
      TypeMappingRegistry registry = service.getTypeMappingRegistry();
      TypeMapping mapping = registry.createTypeMapping();
      registerBeanMapping(mapping, Account.class, "Account");
      registry.register("http://schemas.xmlsoap.org/soap/encoding/", mapping);
   }

   protected void registerBeanMapping(TypeMapping mapping, Class type, String name) {
      QName qName = new QName("http://localhost:8080/account/services/accountService", name);
      mapping.register(type, qName,
          new BeanSerializerFactory(type, qName),
          new BeanDeserializerFactory(type, qName));
   }
}
```

17.5.4. Registering our own Handler

In this section we will register our own `javax.rpc.xml.handler.Handler` to the web service proxy where we can do custom code before the SOAP message is sent over the wire. The `Handler` is a callback interface. There is a convenience base class provided in `jaxrpc.jar`, namely `javax.rpc.xml.handler.GenericHandler` that we will extend:

```
public class AccountHandler extends GenericHandler {

    public QName[] getHeaders() {
        return null;
    }

    public boolean handleRequest(MessageContext context) {
        SOAPMessageContext smc = (SOAPMessageContext) context;
        SOAPMessage msg = smc.getMessage();
```

```
        try {
            SOAPEnvelope envelope = msg.getSOAPPart().getEnvelope();
            SOAPHeader header = envelope.getHeader();
            // ...

        } catch (SOAPException e) {
            throw new JAXRPCException(e);
        }

        return true;
    }
}
```

What we need to do now is to register our AccountHandler to JAX-RPC Service so it would invoke `handleRequest(..)` before the message is sent over the wire. Spring has at this time of writing no declarative support for registering handlers, so we must use the programmatic approach. However Spring has made it very easy for us to do this as we can override the `postProcessJaxRpcService(..)` method that is designed for this:

```
public class AccountHandlerJaxRpcPortProxyFactoryBean extends JaxRpcPortProxyFactoryBean {

    protected void postProcessJaxRpcService(Service service) {
        QName port = new QName(this.getNamespaceUri(), this.getPortName());
        List list = service.getHandlerRegistry().getHandlerChain(port);
        list.add(new HandlerInfo(AccountHandler.class, null, null));

        logger.info("Registered JAX-RPC Handler [" + AccountHandler.class.getName() + "] on port " + port);
    }
}
```

The last thing we must remember to do is to change the Spring configuration to use our factory bean:

```
<bean id="accountWebService" class="example.AccountHandlerJaxRpcPortProxyFactoryBean">
    <!-- ... -->
</bean>
```

17.5.5. Exposing web services using XFire

XFire is a lightweight SOAP library, hosted by Codehaus. Exposing XFire is done using a XFire context that shipping with XFire itself in combination with a RemoteExporter-style bean you have to add to your `WebApplicationContext`. As with all methods that allow you to expose service, you have to create a `DispatcherServlet` with a corresponding `WebApplicationContext` containing the services you will be exposing:

```
<servlet>
  <servlet-name>xfire</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
```

You also have to link in the XFire configuration. This is done by adding a context file to the `contextConfigLocations` context parameter picked up by the `ContextLoaderListener` (or `ContextLoaderServlet` for that matter).

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath:org/codehaus/xfire/spring/xfire.xml
  </param-value>
</context-param>

<listener>
  <listener-class>
```

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

After you added a servlet mapping (mapping `/*` to the XFire servlet declared above) you only have to add one extra bean to expose the service using XFire. Add for example the following configuration in your `'xfire-servlet.xml'` file:

```
<beans>

  <bean name="/Echo" class="org.codehaus.xfire.spring.remoting.XFireExporter">
    <property name="serviceInterface" value="org.codehaus.xfire.spring.Echo"/>
    <property name="serviceBean">
            <bean class="org.codehaus.xfire.spring.EchoImpl"/>
    </property>
    <!-- the XFire bean is defined in the xfire.xml file -->
    <property name="xfire" ref="xfire"/>
  </bean>

</beans>
```

XFire handles the rest. It introspects your service interface and generates a WSDL from it. Parts of this documentation have been taken from the XFire site; for more detailed information on XFire Spring integration, navigate to http://docs.codehaus.org/display/XFIRE/Spring.

17.6. JMS

It is also possible to expose services transparently using JMS as the underlying communication protocol. The JMS remoting support in the Spring Framework is pretty basic - it sends and receives on the `same thread` and in the *same non-transactional* `Session`, and as such throughput will be very implementation dependent.

The following interface is used on both the server and the client side.

```
package com.foo;

public interface CheckingAccountService {

   void cancelAccount(Long accountId);
}
```

The following simple implementation of the above interface is used on the server-side.

```
package com.foo;

public class SimpleCheckingAccountService implements CheckingAccountService {

   public void cancelAccount(Long accountId) {
      System.out.println("Cancelling account [" + accountId + "]");
   }
}
```

This configuration file contains the JMS-infrastructure beans that are shared on both the client and server.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <bean id="connectionFactory" class="org.apache.activemq.ActiveMQConnectionFactory">
      <property name="brokerURL" value="tcp://ep-t43:61616"/>
   </bean>
```

```
   <bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
      <constructor-arg value="mmm"/>
   </bean>

</beans>
```

17.6.1. Server-side configuration

On the server, you just need to expose the service object using the `JmsInvokerServiceExporter`.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <bean id="checkingAccountService"
        class="org.springframework.jms.remoting.JmsInvokerServiceExporter">
      <property name="serviceInterface" value="com.foo.CheckingAccountService"/>
      <property name="service">
         <bean class="com.foo.SimpleCheckingAccountService"/>
      </property>
   </bean>

   <bean class="org.springframework.jms.listener.SimpleMessageListenerContainer">
      <property name="connectionFactory" ref="connectionFactory"/>
      <property name="destination" ref="queue"/>
      <property name="concurrentConsumers" value="3"/>
      <property name="messageListener" ref="checkingAccountService"/>
   </bean>

</beans>
```

```
package com.foo;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Server {

   public static void main(String[] args) throws Exception {
      new ClassPathXmlApplicationContext(new String[]{"com/foo/server.xml", "com/foo/jms.xml"});
   }
}
```

17.6.2. Client-side configuration

The client merely needs to create a client-side proxy that will implement the agreed upon interface (`CheckingAccountService`). The resulting object created off the back of the following bean definition can be injected into other client side objects, and the proxy will take care of forwarding the call to the server-side object via JMS.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <bean id="checkingAccountService"
        class="org.springframework.jms.remoting.JmsInvokerProxyFactoryBean">
      <property name="serviceInterface" value="com.foo.CheckingAccountService"/>
      <property name="connectionFactory" ref="connectionFactory"/>
      <property name="queue" ref="queue"/>
   </bean>

</beans>
```

```
package com.foo;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Client {

   public static void main(String[] args) throws Exception {
      ApplicationContext ctx = new ClassPathXmlApplicationContext(
               new String[]{"com/foo/client.xml", "com/foo/jms.xml"});
      CheckingAccountService service = (CheckingAccountService) ctx.getBean("checkingAccountService");
      service.cancelAccount(new Long(10));
   }
}
```

You may also wish to investigate the support provided by the Lingo project, which (to quote the homepage blurb) "*... is a lightweight POJO based remoting and messaging library based on the Spring Framework's remoting libraries which extends it to support JMS.*"

17.7. Auto-detection is not implemented for remote interfaces

The main reason why auto-detection of implemented interfaces does not occur for remote interfaces is to avoid opening too many doors to remote callers. The target object might implement internal callback interfaces like `InitializingBean` or `DisposableBean` which one would not want to expose to callers.

Offering a proxy with all interfaces implemented by the target usually does not matter in the local case. But when exporting a remote service, you should expose a specific service interface, with specific operations intended for remote usage. Besides internal callback interfaces, the target might implement multiple business interfaces, with just one of them intended for remote exposure. For these reasons, we *require* such a service interface to be specified.

This is a trade-off between configuration convenience and the risk of accidental exposure of internal methods. Always specifying a service interface is not too much effort, and puts you on the safe side regarding controlled exposure of specific methods.

17.8. Considerations when choosing a technology

Each and every technology presented here has its drawbacks. You should carefully consider you needs, the services your exposing and the objects you'll be sending over the wire when choosing a technology.

When using RMI, it's not possible to access the objects through the HTTP protocol, unless you're tunneling the RMI traffic. RMI is a fairly heavy-weight protocol in that it support full-object serialization which is important when using a complex data model that needs serialization over the wire. However, RMI-JRMP is tied to Java clients: It is a Java-to-Java remoting solution.

Spring's HTTP invoker is a good choice if you need HTTP-based remoting but also rely on Java serialization. It shares the basic infrastructure with RMI invokers, just using HTTP as transport. Note that HTTP invokers are not only limited to Java-to-Java remoting but also to Spring on both the client and server side. (The latter also applies to Spring's RMI invoker for non-RMI interfaces.)

Hessian and/or Burlap might provide significant value when operating in a heterogeneous environment, because they explicitly allow for non-Java clients. However, non-Java support is still limited. Known issues include the serialization of Hibernate objects in combination with lazily-initialized collections. If you have such a data model, consider using RMI or HTTP invokers instead of Hessian.

JMS can be useful for providing clusters of services and allowing the JMS broker to take care of load balancing, discovery and auto-failover. By default Java serialization is used when using JMS remoting but the JMS provider could use a different mechanism for the wire formatting, such as XStream to allow servers to be implemented in other technologies.

Last but not least, EJB has an advantage over RMI in that it supports standard role-based authentication and authorization and remote transaction propagation. It is possible to get RMI invokers or HTTP invokers to support security context propagation as well, although this is not provided by core Spring: There are just appropriate hooks for plugging in third-party or custom solutions here.

Chapter 18. Enterprise Java Bean (EJB) integration

18.1. Introduction

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 old Java object) variants, without the 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.

18.2. Accessing EJBs

18.2.1. 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 container, 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 are actually adding real value in such code.

18.2.2. 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 use 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 old 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 an 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 container, we can (in the same context) configure a `LocalStatelessSessionProxyFactoryBean` instance, which will be the 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"/>
  <property name="businessInterface" value="com.mycom.MyComponent"/>
</bean>

<bean id="myController" class="com.mycom.myController">
  <property name="myComponent" ref="myComponent"/>
</bean>
```

There's a lot of work 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 `classname` method on the local EJB and invokes the corresponding business method on the EJB.

The `myController` bean definition sets the `myComponent` property of the controller class to the EJB 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. Additionally, 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 is typically 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 to 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 various 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 then 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`.

18.2.3. 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 written 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 exceptions, 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 re-thrown 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.

18.3. Using Spring's convenience EJB implementation classes

Spring also provides convenience classes to help you implement EJBs. These are designed to encourage the good practice 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 a Message Driven bean, you need only derive your implementation class from `AbstractStatelessSessionBean`, `AbstractStatefulSessionBean`, and `AbstractMessageDrivenBean`/`AbstractJmsMessageDrivenBean`, respectively.

Consider an example Stateless Session bean which actually delegates the implementation to a plain java service object. We have the business interface:

```
public interface MyComponent {
    public void myMethod(...);
    ...
}
```

We also have the plain Java implementation object:

```
public class MyComponentImpl implements MyComponent {
    public String myMethod(...) {
        ...
    }
    ...
}
```

And finally the Stateless Session Bean itself:

```
public class MyComponentEJB 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 Spring IoC container 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 overridden 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 container 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 behavior of the `ContextJndiBeanFactoryLocator` classes which is 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 initialization 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 user may want to override the default `ContextJndiBeanFactoryLocator` usage and use another `BeanFactoryLocator` variant, such as the `ContextSingletonBeanFactoryLocator` which can load and use a shared container 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);
```

```
   }
```

You would then need to create a bean definition file named `beanRefContext.xml`. This file defines all bean factories (usually in the form of application contexts) that may be used in the EJB. In many cases, this file will only contain a single bean definition such as this (where `businessApplicationContext.xml` contains the bean definitions for all business service POJOs):

```
<beans>
    <bean id="businessBeanFactory" class="org.springframework.context.support.ClassPathXmlApplicationContext">
        <constructor-arg value="businessApplicationContext.xml" />
    </bean>
</beans>
```

In the above example, the `ServicesConstants.PRIMARY_CONTEXT_ID` constant would be defined as follows:

```
public static final String ServicesConstants.PRIMARY_CONTEXT_ID = "businessBeanFactory";
```

Please see the respective Javadocs for the `BeanFactoryLocator` and `ContextSingletonBeanFactoryLocator` classes for more information on their usage.

Chapter 19. JMS

19.1. Introduction

Spring provides a JMS abstraction framework that simplifies the use of the JMS API and shields the user from differences between the JMS 1.0.2 and 1.1 APIs.

JMS can be roughly divided into two areas of functionality, namely the production and consumption of messages. The `JmsTemplate` class is used for message production and synchronous message reception. For asynchronous reception similar to J2EE's message-driven bean style, Spring provides a number of message listener containers that are used to create Message-Driven POJOs (MDPs).

Domain Unification

There are two major releases of the JMS specification, 1.0.2 and 1.1.

JMS 1.0.2 defined two types of messaging domains, point-to-point (Queues) and publish/subscribe (Topics). The 1.0.2 API reflected these two messaging domains by providing a parallel class hierarchy for each domain. As a result, a client application became domain specific in its use of the JMS API. JMS 1.1 introduced the concept of domain unification that minimized both the functional differences and client API differences between the two domains. As an example of a functional difference that was removed, if you use a JMS 1.1 provider you can transactionally consume a message from one domain and produce a message on the other using the same `Session`.

Note: The JMS 1.1 specification was released in April 2002 and incorporated as part of J2EE 1.4 in November 2003. As a result, common J2EE 1.3 application servers which are still in widespread use (such as BEA WebLogic 8.1 and IBM WebSphere 5.1) are based on JMS 1.0.2.

The package `org.springframework.jms.core` provides the core functionality for using JMS. It contains JMS template classes that simplifies the use of the JMS by handling the creation and release of resources, much like the `JdbcTemplate` does for JDBC. The design principle common to Spring template classes is to provide helper methods to perform common operations and for more sophisticated usage, delegate the essence of the processing task to user implemented callback interfaces. The JMS template follows the same design. The classes offer various convenience methods for the sending of messages, consuming a message synchronously, and exposing the JMS session and message producer to the user.

The package `org.springframework.jms.support` provides JMSException translation functionality. The translation converts the checked `JMSException` hierarchy to a mirrored hierarchy of unchecked exceptions. If there are any provider specific subclasses of the checked `javax.jms.JMSException`, this exception is wrapped in the unchecked `UncategorizedJmsException`.

The package `org.springframework.jms.support.converter` provides a `MessageConverter` abstraction to convert between Java objects and JMS messages.

The package `org.springframework.jms.support.destination` provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI.

Finally, the package `org.springframework.jms.connection` provides an implementation of the `ConnectionFactory` suitable for use in standalone applications. It also contains an implementation of Spring's `PlatformTransactionManager` for JMS (the cunningly named `JmsTransactionManager`). This allows for

seamless integration of JMS as a transactional resource into Spring's transaction management mechanisms.

19.2. Using Spring JMS

19.2.1. `JmsTemplate`

There are two variants of the functionality offered by the `JmsTemplate`: the `JmsTemplate` uses the JMS 1.1 API, and the subclass `JmsTemplate102` uses the JMS 1.0.2 API.

Code that uses the `JmsTemplate` only needs to implement callback interfaces giving them a clearly defined contract. The `MessageCreator` callback interface creates a message given a `Session` provided by the calling code in `JmsTemplate`. In order to allow for more complex usage of the JMS API, the callback `SessionCallback` provides the user with the JMS session and the callback `ProducerCallback` exposes a `Session` and `MessageProducer` pair.

The JMS API exposes two types of send methods, one that takes delivery mode, priority, and time-to-live as Quality of Service (QOS) parameters and one that takes no QOS parameters which uses default values. Since there are many send methods in `JmsTemplate`, the setting of the QOS parameters have been exposed as bean properties to avoid duplication in the number of send methods. Similarly, the timeout value for synchronous receive calls is set using the property `setReceiveTimeout`.

Some JMS providers allow the setting of default QOS values administratively through the configuration of the ConnectionFactory. This has the effect that a call to `MessageProducer`'s send method `send(Destination destination, Message message)` will use different QOS default values than those specified in the JMS specification. In order to provide consistent management of QOS values, the `JmsTemplate` must therefore be specifically enabled to use its own QOS values by setting the boolean property isExplicitQosEnabled to `true`.

Note

Instances of the `JmsTemplate` class are *thread-safe once configured*. This is important because it means that you can configure a single instance of a `JmsTemplate` and then safely inject this *shared* reference into multiple collaborators. To be clear, the `JmsTemplate` is stateful, in that it maintains a reference to a `ConnectionFactory`, but this state is *not* conversational state.

19.2.2. Connections

The `JmsTemplate` requires a reference to a `ConnectionFactory`. The `ConnectionFactory` is part of the JMS specification and serves as the entry point for working with JMS. It is used by the client application as a factory to create connections with the JMS provider and encapsulates various configuration parameters, many of which are vendor specific such as SSL configuration options.

When using JMS inside an EJB, the vendor provides implementations of the JMS interfaces so that they can participate in declarative transaction management and perform pooling of connections and session. In order to use this implementation, J2EE containers typically require that you declare a JMS connection factory as a resource-ref inside the EJB or servlet deployment descriptors. To ensure the use of these features with the `JmsTemplate` inside an EJB, the client application should ensure that it references the managed implementation of the `ConnectionFactory`.

Spring provides an implementation of the `ConnectionFactory` interface, `SingleConnectionFactory`, that will return the same `Connection` on all `createConnection` calls and ignore calls to `close`. This is useful for testing and standalone environments so that the same connection can be used for multiple `JmsTemplate` calls that may

span any number of transactions. `SingleConnectionFactory` takes a reference to a standard `ConnectionFactory` that would typically come from JNDI.

19.2.3. Destination Management

Destinations, like ConnectionFactories, are JMS administered objects that can be stored and retrieved in JNDI. When configuring a Spring application context you can use the JNDI factory class `JndiObjectFactoryBean` to perform dependency injection on your object's references to JMS destinations. However, often this strategy is cumbersome if there are a large number of destinations in the application or if there are advanced destination management features unique to the JMS provider. Examples of such advanced destination management would be the creation of dynamic destinations or support for a hierarchical namespace of destinations. The `JmsTemplate` delegates the resolution of a destination name to a JMS destination object to an implementation of the interface `DestinationResolver`. `DynamicDestinationResolver` is the default implementation used by `JmsTemplate` and accommodates resolving dynamic destinations. A `JndiDestinationResolver` is also provided that acts as a service locator for destinations contained in JNDI and optionally falls back to the behavior contained in `DynamicDestinationResolver`.

Quite often the destinations used in a JMS application are only known at runtime and therefore cannot be administratively created when the application is deployed. This is often because there is shared application logic between interacting system components that create destinations at runtime according to a well-known naming convention. Even though the creation of dynamic destinations are not part of the JMS specification, most vendors have provided this functionality. Dynamic destinations are created with a name defined by the user which differentiates them from temporary destinations and are often not registered in JNDI. The API used to create dynamic destinations varies from provider to provider since the properties associated with the destination are vendor specific. However, a simple implementation choice that is sometimes made by vendors is to disregard the warnings in the JMS specification and to use the `TopicSession` method `createTopic(String topicName)` or the `QueueSession` method `createQueue(String queueName)` to create a new destination with default destination properties. Depending on the vendor implementation, `DynamicDestinationResolver` may then also create a physical destination instead of only resolving one.

The boolean property pubSubDomain is used to configure the `JmsTemplate` with knowledge of what JMS domain is being used. By default the value of this property is false, indicating that the point-to-point domain, Queues, will be used. In the 1.0.2 implementation the value of this property determines if the `JmsTemplate`'s send operations will send a message to a `Queue` or to a `Topic`. This flag has no effect on send operations for the 1.1 implementation. However, in both implementations, this property determines the behavior of dynamic destination resolution via implementations of the `DestinationResolver` interface.

You can also configure the `JmsTemplate` with a default destination via the property defaultDestination. The default destination will be used with send and receive operations that do not refer to a specific destination.

19.2.4. Message Listener Containers

One of the most common uses of JMS messages in the EJB world is to drive message-driven beans (MDBs). Spring offers a solution to create message-driven POJOs (MDPs) in a way that does not tie a user to an EJB container. (See the section entitled Section 19.4.2, “Asynchronous Reception - Message-Driven POJOs” for detailed coverage of Spring's MDP support.)

A message listener container is used to receive messages from a JMS message queue and drive the MessageListener that is injected into it. The listener container is responsible for all threading of message reception and dispatches into the listener for processing. A message listener container is the intermediary between an MDP and a messaging provider, and takes care of registering to receive messages, participating in transactions, resource acquisition and release, exception conversion and suchlike. This allows you as an

application developer to write the (possibly complex) business logic associated with receiving a message (and possibly responding to it), and delegates boilerplate JMS infrastructure concerns to the framework.

There are three standard JMS message listener containers packaged with Spring, each with its specialised feature set.

19.2.4.1. `SimpleMessageListenerContainer`

This message listener container is the simplest of the three standard flavors. It simply creates a fixed number of JMS sessions at startup and uses them throughout the lifespan of the container. This container doesn't allow for dynamic adaption to runtime demands or participate in externally managed transactions. However, it does have the fewest requirements on the JMS provider: This listener container only requires simple JMS API compliance.

19.2.4.2. `DefaultMessageListenerContainer`

This message listener container is the one used in most cases. In contrast to `SimpleMessageListenerContainer`, this container variant does allow for dynamic adaption to runtime demands and is able to participate in externally managed transactions. Each received message is registered with an XA transaction (when configured with a `JtaTransactionManager`); processing can take advantage of XA transation semantics. This listener container strikes a good balance between low requirements on the JMS provider and good functionality including transaction participation.

19.2.4.3. `ServerSessionMessageListenerContainer`

This listener container leverages the JMS ServerSessionPool SPI to allow for dynamic management of JMS sessions. The use of this variety of message listener container enables the provider to perform dynamic runtime tuning but, at the expense of requiring the JMS provider to support the ServerSessionPool SPI. If there is no need for provider-driven runtime tuning, look at the `DefaultMessageListenerContainer` or the `SimpleMessageListenerContainer` instead.

19.2.5. Transaction management

Spring provides a `JmsTransactionManager` that manages transactions for a single JMS `ConnectionFactory`. This allows JMS applications to leverage the managed transaction features of Spring as described in Chapter 9, *Transaction management*. The `JmsTransactionManager` performs local resource transactions, binding a JMS Connection/Session pair from the specified `ConnectionFactory` to the thread. `JmsTemplate` automatically detects such transactional resources and operates on them accordingly.

In a J2EE environment, the `ConnectionFactory` will pool Connections and Sessions, so those resources are efficiently reused across transactions. In a standalone environment, using Spring's `SingleConnectionFactory` will result in a shared JMS `Connection`, with each transaction having its own independent `Session`. Alternatively, consider the use of a provider-specific pooling adapter such as ActiveMQ's `PooledConnectionFactory` class.

`JmsTemplate` can also be used with the `JtaTransactionManager` and an XA-capable JMS `ConnectionFactory` for performing distributed transactions. Note that this requires the use of a JTA transaction manager as well as a properly XA-configured ConnectionFactory! (Check your J2EE server's / JMS provider's documentation.)

Reusing code across a managed and unmanaged transactional environment can be confusing when using the JMS API to create a `Session` from a `Connection`. This is because the JMS API has only one factory method to create a `Session` and it requires values for the transaction and acknowledgement modes. In a managed

environment, setting these values is the responsibility of the environment's transactional infrastructure, so these values are ignored by the vendor's wrapper to the JMS Connection. When using the `JmsTemplate` in an unmanaged environment you can specify these values through the use of the properties `sessionTransacted` and `sessionAcknowledgeMode`. When using a `PlatformTransactionManager` with `JmsTemplate`, the template will always be given a transactional JMS `Session`.

19.3. Sending a `Message`

The `JmsTemplate` contains many convenience methods to send a message. There are send methods that specify the destination using a `javax.jms.Destination` object and those that specify the destination using a string for use in a JNDI lookup. The send method that takes no destination argument uses the default destination. Here is an example that sends a message to a queue using the 1.0.2 implementation.

```
import javax.jms.ConnectionFactory;
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Queue;
import javax.jms.Session;

import org.springframework.jms.core.MessageCreator;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.JmsTemplate102;

public class JmsQueueSender {

  private JmsTemplate jmsTemplate;
  private Queue queue;

  public void setConnectionFactory(ConnectionFactory cf) {
    this.jmsTemplate = new JmsTemplate102(cf, false);
  }

  public void setQueue(Queue queue) {
    this.queue = queue;
  }

  public void simpleSend() {
    this.jmsTemplate.send(this.queue, new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        return session.createTextMessage("hello queue world");
      }
    });
  }
}
```

This example uses the `MessageCreator` callback to create a text message from the supplied `Session` object and the `JmsTemplate` is constructed by passing a reference to a `ConnectionFactory` and a boolean specifying the messaging domain. A zero argument constructor and connectionFactory / queue bean properties are provided and can be used for constructing the instance (using a BeanFactory or plain Java code). Alternatively, consider deriving from Spring's `JmsGatewaySupport` convenience base class, which provides pre-built bean properties for JMS configuration.

When configuring the JMS 1.0.2 support in an application context, it is important to remember setting the value of the boolean property pubSubDomain property in order to indicate if you want to send to Queues or Topics.

The method `send(String destinationName, MessageCreator creator)` lets you send to a message using the string name of the destination. If these names are registered in JNDI, you should set the destinationResolver property of the template to an instance of `JndiDestinationResolver`.

If you created the `JmsTemplate` and specified a default destination, the `send(MessageCreator c)` sends a message to that destination.

19.3.1. Using Message Converters

In order to facilitate the sending of domain model objects, the `JmsTemplate` has various send methods that take a Java object as an argument for a message's data content. The overloaded methods `convertAndSend` and `receiveAndConvert` in `JmsTemplate` delegate the conversion process to an instance of the `MessageConverter` interface. This interface defines a simple contract to convert between Java objects and JMS messages. The default implementation `SimpleMessageConverter` supports conversion between `String` and `TextMessage`, `byte[]` and `BytesMesssage`, and `java.util.Map` and `MapMessage`. By using the converter, you and your application code can focus on the business object that is being sent or received via JMS and not be concerned with the details of how it is represented as a JMS message.

The sandbox currently includes a `MapMessageConverter` which uses reflection to convert between a JavaBean and a `MapMessage`. Other popular implementations choices you might implement yourself are Converters that use an existing XML marshalling package, such as JAXB, Castor, XMLBeans, or XStream, to create a `TextMessage` representing the object.

To accommodate the setting of a message's properties, headers, and body that can not be generically encapsulated inside a converter class, the `MessagePostProcessor` interface gives you access to the message after it has been converted, but before it is sent. The example below demonstrates how to modify a message header and a property after a `java.util.Map` is converted to a message.

```
public void sendWithConversion() {
  Map m = new HashMap();
  m.put("Name", "Mark");
  m.put("Age", new Integer(47));
  jmsTemplate.convertAndSend("testQueue", m, new MessagePostProcessor() {
    public Message postProcessMessage(Message message) throws JMSException {
      message.setIntProperty("AccountID", 1234);
      message.setJMSCorrelationID("123-00001");
      return message;
    }
  });
}
```

This results in a message of the form:

```
MapMessage={
  Header={
    ... standard headers ...
    CorrelationID={123-00001}
  }
  Properties={
    AccountID={Integer:1234}
  }
  Fields={
    Name={String:Mark}
    Age={Integer:47}
  }
}
```

19.3.2. `SessionCallback` and `ProducerCallback`

While the send operations cover many common usage scenarios, there are cases when you want to perform multiple operations on a JMS `Session` or `MessageProducer`. The `SessionCallback` and `ProducerCallback` expose the JMS `Session` and `Session` / `MessageProducer` pair respectfully. The `execute()` methods on `JmsTemplate` execute these callback methods.

19.4. Receiving a message

19.4.1. Synchronous Reception

While JMS is typically associated with asynchronous processing, it is possible to consume messages synchronously. The overloaded `receive(..)` methods provide this functionality. During a synchronous receive, the calling thread blocks until a message becomes available. This can be a dangerous operation since the calling thread can potentially be blocked indefinitely. The property receiveTimeout specifies how long the receiver should wait before giving up waiting for a message.

19.4.2. Asynchronous Reception - Message-Driven POJOs

In a fashion similar to a Message-Driven Bean (MDB) in the EJB world, the Message-Driven POJO (MDP) acts as a receiver for JMS messages. The one restriction (but see also below for the discussion of the `MessageListenerAdapter` class) on an MDP is that it must implement the `javax.jms.MessageListener` interface. Please also be aware that in the case where your POJO will be receiving messages on multiple threads, it is important to ensure that your implementation is thread-safe.

Below is a simple implementation of an MDP:

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

public class ExampleListener implements MessageListener {

  public void onMessage(Message message) {
    if (message instanceof TextMessage) {
      try {
        System.out.println(((TextMessage) message).getText());
      } catch (JMSException ex) {
        throw new RuntimeException(ex);
      }
    } else {
      throw new IllegalArgumentException("Message must be of type TextMessage");
    }
  }
}
```

Once you've implemented your `MessageListener`, it's time to create a message listener container.

Find below an example of how to define and configure one of the message listener containers that ships with Spring (in this case the `DefaultMessageListenerContainer`).

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener" />

<!-- and this is the message listener container -->
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="destination" ref="destination"/>
  <property name="messageListener" ref="messageListener" />
</bean>
```

Please refer to the Spring Javadoc of the various message listener containers for a full description of the features supported by each implementation.

19.4.3. The `SessionAwareMessageListener` interface

The `SessionAwareMessageListener` interface is a Spring-specific interface that provides a similar contract the JMS `MessageListener` interface, but also provides the message handling method with access to the JMS

`Session` from which the `Message` was received.

```
package org.springframework.jms.listener;

public interface SessionAwareMessageListener {

    void onMessage(Message message, Session session) throws JMSException;
}
```

You can choose to have your MDPs implement this interface (in preference to the standard JMS `MessageListener` interface) if you want your MDPs to be able to respond to any received messages (using the `Session` supplied in the `onMessage(Message, Session)` method). All of the message listener container implementations that ship wth Spring have support for MDPs that implement either the `MessageListener` or `SessionAwareMessageListener` interface. Classes that implement the `SessionAwareMessageListener` come with the caveat that they are then tied to Spring through the interface. The choice of whether or not to use it is left entirely up to you as an application developer or architect.

Please note that the `'onMessage(..)'` method of the `SessionAwareMessageListener` interface throws `JMSException`. In contrast to the standard JMS `MessageListener` interface, when using the `SessionAwareMessageListener` interface, it is the responsibility of the client code to handle any exceptions thrown.

19.4.4. The `MessageListenerAdapter`

The `MessageListenerAdapter` class is the final component in Spring's asynchronous messaging support: in a nutshell, it allows you to expose almost *any* class as a MDP (there are of course some constraints).

Note

If you are using the JMS 1.0.2 API, you will want to use the `MessageListenerAdapter102` class which provides the exact same functionality and value add as the `MessageListenerAdapter` class, but for the JMS 1.0.2 API.

Consider the following interface definition. Notice that although the interface extends neither the `MessageListener` nor `SessionAwareMessageListener` interfaces, it can still be used as a MDP via the use of the `MessageListenerAdapter` class. Notice also how the various message handling methods are strongly typed according to the *contents* of the various `Message` types that they can receive and handle.

```
public interface MessageDelegate {

    void handleMessage(String message);

    void handleMessage(Map message);

    void handleMessage(byte[] message);

    void handleMessage(Serializable message);
}
```

```
public class DefaultMessageDelegate implements MessageDelegate {
    // implementation elided for clarity...
}
```

In particular, note how the above implementation of the `MessageDelegate` interface (the above `DefaultMessageDelegate` class) has *no* JMS dependencies at all. It truly is a POJO that we will make into an MDP via the following configuration.

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultMessageDelegate"/>
    </constructor-arg>
</bean>

<!-- and this is the message listener container... -->
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="destination" ref="destination"/>
  <property name="messageListener" ref="messageListener" />
</bean>
```

Below is an example of another MDP that can only handle the receiving of JMS `TextMessage` messages. Notice how the message handling method is actually called `'receive'` (the name of the message handling method in a `MessageListenerAdapter` defaults to `'handleMessage'`), but it is configurable (as you will see below). Notice also how the `'receive(..)'` method is strongly typed to receive and respond only to JMS `TextMessage` messages.

```
public interface TextMessageDelegate {

    void receive(TextMessage message);
}
```

```
public class DefaultTextMessageDelegate implements TextMessageDelegate {
    // implementation elided for clarity...
}
```

The configuration of the attendant `MessageListenerAdapter` would look like this:

```
<bean id="messageListener" class="org.springframework.jms.listener.adapter.MessageListenerAdapter">
    <constructor-arg>
        <bean class="jmsexample.DefaultTextMessageDelegate"/>
    </constructor-arg>
    <property name="defaultListenerMethod" value="receive"/>
    <!-- we don't want automatic message context extraction -->
    <property name="messageConverter">
        <null/>
    </property>
</bean>
```

Please note that if the above `'messageListener'` receives a JMS `Message` of a type other than `TextMessage`, an `IllegalStateException` will be thrown (and subsequently swallowed). Another of the capabilities of the `MessageListenerAdapter` class is the ability to automatically send back a response `Message` if a handler method returns a non-void value. Consider the interface and class:

```
public interface ResponsiveTextMessageDelegate {

    // notice the return type...
    String receive(TextMessage message);
}
```

```
public class DefaultResponsiveTextMessageDelegate implements ResponsiveTextMessageDelegate {
    // implementation elided for clarity...
}
```

If the above `DefaultResponsiveTextMessageDelegate` is used in conjunction with a `MessageListenerAdapter` then any non-null value that is returned from the execution of the `'receive(..)'` method will (in the default configuration) be converted into a `TextMessage`. The resulting `TextMessage` will then be sent to the `Destination` (if one exists) defined in the JMS Reply-To property of the original `Message`, or the default

`Destination` set on the `MessageListenerAdapter` (if one has been configured); if no `Destination` is found then an `InvalidDestinationException` will be thrown (and please note that this exception *will not* be swallowed and *will* propagate up the call stack).

19.4.5. Processing messages within transactions

Invoking a message listener within a transaction only requires reconfiguration of the listener container.

Local resource transactions can simply be activated through the `sessionTransacted` flag on the listener container definition. Each message listener invocation will then operate within an active JMS transaction, with message reception rolled back in case of listener execution failure. Sending a response message (via `SessionAwareMessageListener`) will be part of the same local transaction, but any other resource operations (such as database access) will operate independently. This usually requires duplicate message detection in the listener implementation, covering the case where database processing has committed but message processing failed to commit.

```
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="destination" ref="destination"/>
  <property name="messageListener" ref="messageListener"/>
  <property name="sessionTransacted" value="true"/>
</bean>
```

For participating in an externally managed transaction, you will need to configured a transaction manager and use a listener container which supports externally managed transactions: typically `DefaultMessageListenerContainer`.

To configure a message listener container for XA transaction participation, you'll want to configure a `JtaTransactionManager` (which, by default, delegates to the J2EE server's transaction subsystem). Note that the underlying JMS ConnectionFactory needs to be XA-capable and properly registered with your JTA transaction coordinator! (Check your J2EE server's configuration of JNDI resources.) This allows message recepton as well as e.g. database access to be part of the same transaction (with unified commit semantics, at the expense of XA transaction log overhead).

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

Then you just need to add it to our earlier container configuration. The container will take care of the rest.

```
<bean id="jmsContainer" class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="connectionFactory"/>
  <property name="destination" ref="destination"/>
  <property name="messageListener" ref="messageListener"/>
  <property name="transactionManager" ref="transactionManager"/>
</bean>
```

Chapter 20. JMX

20.1. Introduction

The JMX support in Spring provides you with the features to easily and transparently integrate your Spring application into a JMX infrastructure.

> **JMX?**
>
> This chapter is not an introduction to JMX... it doesn't try to explain the motivations of why one might want to use JMX (or indeed what the letters JMX actually stand for). If you are new to JMX, check out the section entitled Section 20.8, “Further Resources” at the end of this chapter.

Specifically, Spring's JMX support provides four core features:

- The automatic registration of *any* Spring bean as a JMX MBean
- A flexible mechanism for controlling the management interface of your beans
- The declarative exposure of MBeans over remote, JSR-160 connectors
- The simple proxying of both local and remote MBean resources

These features are designed to work without coupling your application components to either Spring or JMX interfaces and classes. Indeed, for the most part your application classes need not be aware of either Spring or JMX in order to take advantage of the Spring JMX features.

20.2. Exporting your beans to JMX

The core class in Spring's JMX framework is the `MBeanExporter`. This class is responsible for taking your Spring beans and registering them with a JMX `MBeanServer`. For example, consider the following class:

```
package org.springframework.jmx;

public class JmxTestBean implements IJmxTestBean {

    private String name;
    private int age;
    private boolean isSuperman;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public int add(int x, int y) {
```

```
        return x + y;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }
}
```

To expose the properties and methods of this bean as attributes and operations of an MBean you simply configure an instance of the `MBeanExporter` class in your configuration file and pass in the bean as shown below:

```
<beans>

  <!-- this bean must not be lazily initialized if the exporting is to happen -->
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter" lazy-init="false">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

The pertinent bean definition from the above configuration snippet is the `exporter` bean. The `beans` property tells the `MBeanExporter` exactly which of your beans must be exported to the JMX `MBeanServer`. In the default configuration, the key of each entry in the `beans` `Map` is used as the `ObjectName` for the bean referenced by the corresponding entry value. This behavior can be changed as described in the section entitled Section 20.4, “Controlling the `ObjectName`s for your beans”.

With this configuration the `testBean` bean is exposed as an MBean under the `ObjectName` `bean:name=testBean1`. By default, all *public* properties of the bean are exposed as attributes and all *public* methods (bar those inherited from the `Object` class) are exposed as operations.

20.2.1. Creating an `MBeanServer`

The above configuration assumes that the application is running in an environment that has one (and only one) `MBeanServer` already running. In this case, Spring will attempt to locate the running `MBeanServer` and register your beans with that server (if any). This behavior is useful when your application is running inside a container such as Tomcat or IBM WebSphere that has it's own `MBeanServer`.

However, this approach is of no use in a standalone environment, or when running inside a container that does not provide an `MBeanServer`. To address this you can create an `MBeanServer` instance declaratively by adding an instance of the `org.springframework.jmx.support.MBeanServerFactoryBean` class to your configuration. You can also ensure that a specific `MBeanServer` is used by setting the value of the `MBeanExporter`'s `server` property to the `MBeanServer` value returned by an `MBeanServerFactoryBean`; for example:

```
<beans>

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"/>

  <!--
    this bean needs to be eagerly pre-instantiated in order for the exporting to occur;
    this means that it must not be marked as lazily initialized
  -->
  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
```

```
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="server" ref="mbeanServer"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

Here an instance of `MBeanServer` is created by the `MBeanServerFactoryBean` and is supplied to the `MBeanExporter` via the server property. When you supply your own `MBeanServer` instance, the `MBeanExporter` will not attempt to locate a running `MBeanServer` and will use the supplied `MBeanServer` instance. For this to work correctly, you must (of course) have a JMX implementation on your classpath.

20.2.2. Reusing an existing `MBeanServer`

If no server is specified, the `MBeanExporter` tries to automatically detect a running `MBeanServer`. This works in most environment where only one `MBeanServer` instance is used, however when multiple instances exist, the exporter might pick the wrong server. In such cases, one should use the `MBeanServer` `agentId` to indicate which instance to be used:

```
<beans>
   <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean">
     <!-- indicate to first look for a server -->
     <property name="locateExistingServerIfPossible" value="true"/>
     <!-- search for the MBeanServer instance with the given agentId -->
     <property name="agentId" value="<MBeanServer instance agentId>"/>
   </bean>

   <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
     <property name="server" ref="mbeanServer"/>
   ...
   </bean>
</beans>
```

For platforms/cases where the existing `MBeanServer` has a dynamic (or unknown) `agentId` which is retrieved through lookup methods, one should use factory-method:

```
<beans>
   <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
     <property name="server">
       <!-- Custom MBeanServerLocator -->
       <bean class="platform.package.MBeanServerLocator" factory-method="locateMBeanServer"/>
     </property>

     <!-- other beans here -->

   </bean>
</beans>
```

20.2.3. Lazy-initialized MBeans

If you configure a bean with the `MBeanExporter` that is also configured for lazy initialization, then the `MBeanExporter` will **not** break this contract and will avoid instantiating the bean. Instead, it will register a proxy with the `MBeanServer` and will defer obtaining the bean from the container until the first invocation on the proxy occurs.

20.2.4. Automatic registration of MBeans

Any beans that are exported through the `MBeanExporter` and are already valid MBeans are registered as-is with the `MBeanServer` without further intervention from Spring. MBeans can be automatically detected by the `MBeanExporter` by setting the `autodetect` property to `true`:

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
  <property name="autodetect" value="true"/>
</bean>

<bean name="spring:mbean=true" class="org.springframework.jmx.export.TestDynamicMBean"/>
```

Here, the bean called `spring:mbean=true` is already a valid JMX MBean and will be automatically registered by Spring. By default, beans that are autodetected for JMX registration have their bean name used as the `ObjectName`. This behavior can be overridden as detailed in the section entitled Section 20.4, “Controlling the `ObjectNames` for your beans”.

20.2.5. Controlling the registration behavior

Consider the scenario where a Spring `MBeanExporter` attempts to register an `MBean` with an `MBeanServer` using the `ObjectName 'bean:name=testBean1'`. If an `MBean` instance has already been registered under that same `ObjectName`, the default behavior is to fail (and throw an `InstanceAlreadyExistsException`).

It is possible to control the behavior of exactly what happens when an `MBean` is registered with an `MBeanServer`. Spring's JMX support allows for three different registration behaviors to control the registration behavior when the registration process finds that an `MBean` has already been registered under the same `ObjectName`; these registration behaviors are summarized on the following table:

Table 20.1. Registration Behaviors

Registration behavior	Explanation
`REGISTRATION_FAIL_ON_EXISTING`	This is the default registration behavior. If an `MBean` instance has already been registered under the same `ObjectName`, the `MBean` that is being registered will not be registered and an `InstanceAlreadyExistsException` will be thrown. The existing `MBean` is unaffected.
`REGISTRATION_IGNORE_EXISTING`	If an `MBean` instance has already been registered under the same `ObjectName`, the `MBean` that is being registered will *not* be registered. The existing `MBean` is unaffected, and no `Exception` will be thrown. This is useful in settings where multiple applications want to share a common `MBean` in a shared `MBeanServer`.
`REGISTRATION_REPLACE_EXISTING`	If an `MBean` instance has already been registered under the same `ObjectName`, the existing `MBean` that was previously registered will be unregistered and the new `MBean` will be registered in it's place (the new `MBean`

Registration behavior	Explanation
	effectively replaces the previous instance).

The above values are defined as constants on the `MBeanRegistrationSupport` class (the `MBeanExporter` class derives from this superclass). If you want to change the default registration behavior, you simply need to set the value of the `registrationBehaviorName` property on your `MBeanExporter` definition to one of those values.

The following example illustrates how to effect a change from the default registration behavior to the `REGISTRATION_REPLACE_EXISTING` behavior:

```
<beans>

    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="beans">
            <map>
                <entry key="bean:name=testBean1" value-ref="testBean"/>
            </map>
        </property>
        <property name="registrationBehaviorName" value="REGISTRATION_REPLACE_EXISTING"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
    </bean>

</beans>
```

20.3. Controlling the management interface of your beans

In the previous example, you had little control over the management interface of your bean; *all* of the *public* properties and methods of each exported bean was exposed as JMX attributes and operations respectively. To exercise finer-grained control over exactly which properties and methods of your exported beans are actually exposed as JMX attributes and operations, Spring JMX provides a comprehensive and extensible mechanism for controlling the management interfaces of your beans.

20.3.1. The `MBeanInfoAssembler` Interface

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `org.springframework.jmx.export.assembler.MBeanInfoAssembler` interface which is responsible for defining the management interface of each bean that is being exposed. The default implementation, `org.springframework.jmx.export.assembler.SimpleReflectiveMBeanInfoAssembler`, simply defines a management interface that exposes all public properties and methods (as you saw in the previous examples). Spring provides two additional implementations of the `MBeanInfoAssembler` interface that allow you to control the generated management interface using either source-level metadata or any arbitrary interface.

20.3.2. Using source-Level metadata

Using the `MetadataMBeanInfoAssembler` you can define the management interfaces for your beans using source level metadata. The reading of metadata is encapsulated by the `org.springframework.jmx.export.metadata.JmxAttributeSource` interface. Out of the box, Spring JMX provides support for two implementations of this interface: `org.springframework.jmx.export.metadata.AttributesJmxAttributeSource` for Commons Attributes and `org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource` for JDK 5.0 annotations.

The `MetadataMBeanInfoAssembler` *must* be configured with an implementation instance of the `JmxAttributeSource` interface for it to function correctly (there is *no* default). For the following example, we will use the Commons Attributes metadata approach.

To mark a bean for export to JMX, you should annotate the bean class with the `ManagedResource` attribute. In the case of the Commons Attributes metadata approach this class can be found in the `org.springframework.jmx.metadata` package. Each method you wish to expose as an operation must be marked with the `ManagedOperation` attribute and each property you wish to expose must be marked with the `ManagedAttribute` attribute. When marking properties you can omit either the annotation of the getter or the setter to create a write-only or read-only attribute respectively.

The example below shows the `JmxTestBean` class that you saw earlier marked with Commons Attributes metadata:

```
package org.springframework.jmx;

/**
 * @@org.springframework.jmx.export.metadata.ManagedResource
 *  (description="My Managed Bean", objectName="spring:bean=test",
 *  log=true, logFile="jmx.log", currencyTimeLimit=15, persistPolicy="OnUpdate",
 *  persistPeriod=200, persistLocation="foo", persistName="bar")
 */
public class JmxTestBean implements IJmxTestBean {

  private String name;

  private int age;


  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *   (description="The Age Attribute", currencyTimeLimit=15)
   */
  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *  (description="The Name Attribute",  currencyTimeLimit=20,
   *   defaultValue="bar", persistPolicy="OnUpdate")
   */
  public void setName(String name) {
    this.name = name;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedAttribute
   *   (defaultValue="foo", persistPeriod=300)
   */
  public String getName() {
    return name;
  }

  /**
   * @@org.springframework.jmx.export.metadata.ManagedOperation
   *  (description="Add Two Numbers Together")
   */
  public int add(int x, int y) {
    return x + y;
  }

  public void dontExposeMe() {
    throw new RuntimeException();
  }
}
```

Here you can see that the `JmxTestBean` class is marked with the `ManagedResource` attribute and that this `ManagedResource` attribute is configured with a set of properties. These properties can be used to configure various aspects of the MBean that is generated by the `MBeanExporter`, and are explained in greater detail later in section entitled Section 20.3.4, “Source-Level Metadata Types”.

You will also notice that both the `age` and `name` properties are annotated with the `ManagedAttribute` attribute, but in the case of the `age` property, only the getter is marked. This will cause both of these properties to be included in the management interface as attributes, but the `age` attribute will be read-only.

Finally, you will notice that the `add(int, int)` method is marked with the `ManagedOperation` attribute whereas the `dontExposeMe()` method is not. This will cause the management interface to contain only one operation, `add(int, int)`, when using the `MetadataMBeanInfoAssembler`.

The code below shows how you configure the `MBeanExporter` to use the `MetadataMBeanInfoAssembler`:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource">
    <property name="attributes">
      <bean class="org.springframework.metadata.commons.CommonsAttributes"/>
    </property>
  </bean>

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>

</beans>
```

Here you can see that an `MetadataMBeanInfoAssembler` bean has been configured with an instance of the `AttributesJmxAttributeSource` class and passed to the `MBeanExporter` through the assembler property. This is all that is required to take advantage of metadata-driven management interfaces for your Spring-exposed MBeans.

20.3.3. Using JDK 5.0 Annotations

To enable the use of JDK 5.0 annotations for management interface definition, Spring provides a set of annotations that mirror the Commons Attribute attribute classes and an implementation of the `JmxAttributeSource` strategy interface, the `AnnotationsJmxAttributeSource` class, that allows the `MBeanInfoAssembler` to read them.

The example below shows a bean where the management interface is defined by the presence of JDK 5.0 annotation types:

```
package org.springframework.jmx;

import org.springframework.jmx.export.annotation.ManagedResource;
import org.springframework.jmx.export.annotation.ManagedOperation;
```

```
import org.springframework.jmx.export.annotation.ManagedAttribute;

@ManagedResource(objectName="bean:name=testBean4", description="My Managed Bean", log=true,
    logFile="jmx.log", currencyTimeLimit=15, persistPolicy="OnUpdate", persistPeriod=200,
    persistLocation="foo", persistName="bar")
public class AnnotationTestBean implements IJmxTestBean {

  private String name;
  private int age;

  @ManagedAttribute(description="The Age Attribute", currencyTimeLimit=15)
  public int getAge() {
    return age;
  }

  public void setAge(int age) {
    this.age = age;
  }

  @ManagedAttribute(description="The Name Attribute",
      currencyTimeLimit=20,
      defaultValue="bar",
      persistPolicy="OnUpdate")
  public void setName(String name) {
    this.name = name;
  }

  @ManagedAttribute(defaultValue="foo", persistPeriod=300)
  public String getName() {
    return name;
  }

  @ManagedOperation(description="Add two numbers")
  @ManagedOperationParameters({
    @ManagedOperationParameter(name = "x", description = "The first number"),
    @ManagedOperationParameter(name = "y", description = "The second number")})
  public int add(int x, int y) {
    return x + y;
  }

  public void dontExposeMe() {
    throw new RuntimeException();
  }
}
```

As you can see little has changed, other than the basic syntax of the metadata definitions. Behind the scenes this approach is a little slower at startup because the JDK 5.0 annotations are converted into the classes used by Commons Attributes. However, this is only a one-off cost and JDK 5.0 annotations give you the added (and valuable) benefit of compile-time checking.

```
<beans>
    <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
        <property name="assembler" ref="assembler"/>
        <property name="namingStrategy" ref="namingStrategy"/>
        <property name="autodetect" value="true"/>
    </bean>

    <bean id="jmxAttributeSource"
          class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>

    <!-- will create management interface using annotation metadata -->
    <bean id="assembler"
          class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <!-- will pick up the ObjectName from the annotation -->
    <bean id="namingStrategy"
          class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
        <property name="attributeSource" ref="jmxAttributeSource"/>
    </bean>

    <bean id="testBean" class="org.springframework.jmx.AnnotationTestBean">
        <property name="name" value="TEST"/>
        <property name="age" value="100"/>
```

```
    </bean>
</beans>
```

20.3.4. Source-Level Metadata Types

The following source level metadata types are available for use in Spring JMX:

Table 20.2. Source-Level Metadata Types

Purpose	Commons Attributes Attribute	JDK 5.0 Annotation	Attribute / Annotation Type
Mark all instances of a `Class` as JMX managed resources	`ManagedResource`	`@ManagedResource`	Class
Mark a method as a JMX operation	`ManagedOperation`	`@ManagedOperation`	Method
Mark a getter or setter as one half of a JMX attribute	`ManagedAttribute`	`@ManagedAttribute`	Method (only getters and setters)
Define descriptions for operation parameters	`ManagedOperationParameter` and `ManagedOperationParameters`	`@ManagedOperationParameter` and `@ManagedOperationParameters`	Method

The following configuration parameters are available for use on these source-level metadata types:

Table 20.3. Source-Level Metadata Parameters

Parameter	Description	Applies to
`ObjectName`	Used by `MetadataNamingStrategy` to determine the `ObjectName` of a managed resource	`ManagedResource`
`description`	Sets the friendly description of the resource, attribute or operation	`ManagedResource`, `ManagedAttribute`, `ManagedOperation`, `ManagedOperationParameter`
`currencyTimeLimit`	Sets the value of the `currencyTimeLimit` descriptor field	`ManagedResource`, `ManagedAttribute`
`defaultValue`	Sets the value of the `defaultValue` descriptor field	`ManagedAttribute`
`log`	Sets the value of the `log` descriptor field	`ManagedResource`
`logFile`	Sets the value of the `logFile`	`ManagedResource`

Parameter	Description	Applies to
	descriptor field	
persistPolicy	Sets the value of the persistPolicy descriptor field	ManagedResource
persistPeriod	Sets the value of the persistPeriod descriptor field	ManagedResource
persistLocation	Sets the value of the persistLocation descriptor field	ManagedResource
persistName	Sets the value of the persistName descriptor field	ManagedResource
name	Sets the display name of an operation parameter	ManagedOperationParameter
index	Sets the index of an operation parameter	ManagedOperationParameter

20.3.5. The `AutodetectCapableMBeanInfoAssembler` interface

To simply configuration even further, Spring introduces the `AutodetectCapableMBeanInfoAssembler` interface which extends the `MBeanInfoAssembler` interface to add support for autodetection of MBean resources. If you configure the `MBeanExporter` with an instance of `AutodetectCapableMBeanInfoAssembler` then it is allowed to "vote" on the inclusion of beans for exposure to JMX.

Out of the box, the only implementation of the `AutodetectCapableMBeanInfo` interface is the `MetadataMBeanInfoAssembler` which will vote to include any bean which is marked with the `ManagedResource` attribute. The default approach in this case is to use the bean name as the `ObjectName` which results in a configuration like this:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <!-- notice how no 'beans' are explicitly configured here -->
    <property name="autodetect" value="true"/>
    <property name="assembler" ref="assembler"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <!-- (for Commons Attributes-based metadata) -->
  <bean id="attributeSource"
        class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource">
    <property name="attributes">
      <bean class="org.springframework.metadata.commons.CommonsAttributes"/>
    </property>
  </bean>

  <!-- (for Java 5+ annotations-based metadata) -->
  <!--
  <bean id="attributeSource"
        class="org.springframework.jmx.export.annotation.AnnotationJmxAttributeSource"/>
  -->

  <bean id="assembler" class="org.springframework.jmx.export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>
```

```
</beans>
```

Notice that in this configuration no beans are passed to the `MBeanExporter`; however, the `JmxTestBean` will still be registered since it is marked with the `ManagedResource` attribute and the `MetadataMBeanInfoAssembler` detects this and votes to include it. The only problem with this approach is that the name of the `JmxTestBean` now has business meaning. You can address this issue by changing the default behavior for `ObjectName` creation as defined in the section entitled Section 20.4, “Controlling the `ObjectNames` for your beans”.

20.3.6. Defining Management interfaces using Java interfaces

In addition to the `MetadataMBeanInfoAssembler`, Spring also includes the `InterfaceBasedMBeanInfoAssembler` which allows you to constrain the methods and properties that are exposed based on the set of methods defined in a collection of interfaces.

Although the standard mechanism for exposing MBeans is to use interfaces and a simple naming scheme, the `InterfaceBasedMBeanInfoAssembler` extends this functionality by removing the need for naming conventions, allowing you to use more than one interface and removing the need for your beans to implement the MBean interfaces.

Consider this interface that is used to define a management interface for the `JmxTestBean` class that you saw earlier:

```
public interface IJmxTestBean {

  public int add(int x, int y);

  public long myOperation();

  public int getAge();

  public void setAge(int age);

  public void setName(String name);

  public String getName();
}
```

This interface defines the methods and properties that will be exposed as operations and attributes on the JMX MBean. The code below shows how to configure Spring JMX to use this interface as the definition for the management interface:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
        <property name="managedInterfaces">
          <value>org.springframework.jmx.IJmxTestBean</value>
        </property>
      </bean>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>
```

```
</beans>
```

Here you can see that the `InterfaceBasedMBeanInfoAssembler` is configured to use the `IJmxTestBean` interface when constructing the management interface for any bean. It is important to understand that beans processed by the `InterfaceBasedMBeanInfoAssembler` are *not* required to implement the interface used to generate the JMX management interface.

In the case above, the `IJmxTestBean` interface is used to construct all management interfaces for all beans. In many cases this is not the desired behavior and you may want to use different interfaces for different beans. In this case, you can pass `InterfaceBasedMBeanInfoAssembler` a `Properties` instance via the `interfaceMappings` property, where the key of each entry is the bean name and the value of each entry is a comma-separated list of interface names to use for that bean.

If no management interface is specified through either the `managedInterfaces` or `interfaceMappings` properties, then the `InterfaceBasedMBeanInfoAssembler` will reflect on the bean and use all of the interfaces implemented by that bean to create the management interface.

20.3.7. Using `MethodNameBasedMBeanInfoAssembler`

The `MethodNameBasedMBeanInfoAssembler` allows you to specify a list of method names that will be exposed to JMX as attributes and operations. The code below shows a sample configuration for this:

```
<bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5" value-ref="testBean"/>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export.assembler.MethodNameBasedMBeanInfoAssembler">
        <property name="managedMethods">
          <value>add,myOperation,getName,setName,getAge</value>
        </property>
      </bean>
    </property>
</bean>
```

Here you can see that the methods `add` and `myOperation` will be exposed as JMX operations and `getName()`, `setName(String)` and `getAge()` will be exposed as the appropriate half of a JMX attribute. In the code above, the method mappings apply to beans that are exposed to JMX. To control method exposure on a bean-by-bean basis, use the `methodMappings` property of `MethodNameMBeanInfoAssembler` to map bean names to lists of method names.

20.4. Controlling the `ObjectName`s for your beans

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `ObjectNamingStrategy` to obtain `ObjectNames` for each of the beans it is registering. The default implementation, `KeyNamingStrategy`, will, by default, use the key of the `beans Map` as the `ObjectName`. In addition, the `KeyNamingStrategy` can map the key of the `beans Map` to an entry in a `Properties` file (or files) to resolve the `ObjectName`. In addition to the `KeyNamingStrategy`, Spring provides two additional `ObjectNamingStrategy` implementations: the `IdentityNamingStrategy` that builds an `ObjectName` based on the JVM identity of the bean and the `MetadataNamingStrategy` that uses source level metadata to obtain the `ObjectName`.

20.4.1. Reading `ObjectName`s from `Properties`

You can configure your own `KeyNamingStrategy` instance and configure it to read `ObjectNames` from a `Properties` instance rather than use bean key. The `KeyNamingStrategy` will attempt to locate an entry in the `Properties` with a key corresponding to the bean key. If no entry is found or if the `Properties` instance is `null` then the bean key itself is used.

The code below shows a sample configuration for the `KeyNamingStrategy`:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.KeyNamingStrategy">
    <property name="mappings">
      <props>
        <prop key="testBean">bean:name=testBean1</prop>
      </props>
    </property>
    <property name="mappingLocations">
      <value>names1.properties,names2.properties</value>
    </property>
  </bean

</beans>
```

Here an instance of `KeyNamingStrategy` is configured with a `Properties` instance that is merged from the `Properties` instance defined by the mapping property and the properties files located in the paths defined by the mappings property. In this configuration, the `testBean` bean will be given the `ObjectName` `bean:name=testBean1` since this is the entry in the `Properties` instance that has a key corresponding to the bean key.

If no entry in the `Properties` instance can be found then the bean key name is used as the `ObjectName`.

20.4.2. Using the `MetadataNamingStrategy`

The `MetadataNamingStrategy` uses `objectName` property of the `ManagedResource` attribute on each bean to create the `ObjectName`. The code below shows the configuration for the `MetadataNamingStrategy`:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export.naming.MetadataNamingStrategy">
    <property name="attributeSource" ref="attributeSource"/>
```

```
  </bean>

  <bean id="attributeSource"
      class="org.springframework.jmx.export.metadata.AttributesJmxAttributeSource"/>

</beans>
```

20.5. JSR-160 Connectors

For remote access, Spring JMX module offers two `FactoryBean` implementations inside the `org.springframework.jmx.support` package for creating both server- and client-side connectors.

20.5.1. Server-side Connectors

To have Spring JMX create, start and expose a JSR-160 `JMXConnectorServer` use the following configuration:

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean"/>
```

By default `ConnectorServerFactoryBean` creates a `JMXConnectorServer` bound to `"service:jmx:jmxmp://localhost:9875"`. The `serverConnector` bean thus exposes the local `MBeanServer` to clients through the JMXMP protocol on localhost, port 9875. Note that the JMXMP protocol is marked as optional by the JSR 160 specification: currently, the main open-source JMX implementation, MX4J, and the one provided with J2SE 5.0 do *not* support JMXMP.

To specify another URL and register the `JMXConnectorServer` itself with the `MBeanServer` use the `serviceUrl` and `ObjectName` properties respectively:

```
<bean id="serverConnector"
      class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=rmi"/>
  <property name="serviceUrl"
            value="service:jmx:rmi://localhost/jndi/rmi://localhost:1099/myconnector"/>
</bean>
```

If the `ObjectName` property is set Spring will automatically register your connector with the `MBeanServer` under that `ObjectName`. The example below shows the full set of parameters which you can pass to the `ConnectorServerFactoryBean` when creating a JMXConnector:

```
<bean id="serverConnector"
      class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=iiop"/>
  <property name="serviceUrl"
              value="service:jmx:iiop://localhost/jndi/iiop://localhost:900/myconnector"/>
  <property name="threaded" value="true"/>
  <property name="daemon" value="true"/>
  <property name="environment">
    <map>
      <entry key="someKey" value="someValue"/>
    </map>
  </property>
</bean>
```

Note that when using a RMI-based connector you need the lookup service (tnameserv or rmiregistry) to be started in order for the name registration to complete. If you are using Spring to export remote services for you via RMI, then Spring will already have constructed an RMI registry. If not, you can easily start a registry using the following snippet of configuration:

```
<bean id="registry" class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
  <property name="port" value="1099"/>
</bean>
```

20.5.2. Client-side Connectors

To create an `MBeanServerConnection` to a remote JSR-160 enabled `MBeanServer` use the `MBeanServerConnectionFactoryBean` as shown below:

```
<bean id="clientConnector" class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
  <property name="serviceUrl" value="service:jmx:rmi://localhost:9875"/>
</bean>
```

20.5.3. JMX over Burlap/Hessian/SOAP

JSR-160 permits extensions to the way in which communication is done between the client and the server. The examples above are using the mandatory RMI-based implementation required by the JSR-160 specification (IIOP and JRMP) and the (optional) JMXMP. By using other providers or JMX implementations (such as MX4J) you can take advantage of protocols like SOAP, Hessian, Burlap over simple HTTP or SSL and others:

```
<bean id="serverConnector" class="org.springframework.jmx.support.ConnectorServerFactoryBean">
  <property name="objectName" value="connector:name=burlap"/>
  <property name="serviceUrl" value="service:jmx:burlap://localhost:9874"/>
</bean>
```

In the case of the above example, MX4J 3.0.0 was used; see the official MX4J documentation for more information.

20.6. Accessing MBeans via Proxies

Spring JMX allows you to create proxies that re-route calls to MBeans registered in a local or remote `MBeanServer`. These proxies provide you with a standard Java interface through which you can interact with your MBeans. The code below shows how to configure a proxy for an MBean running in a local `MBeanServer`:

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName" value="bean:name=testBean"/>
    <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
</bean>
```

Here you can see that a proxy is created for the MBean registered under the `ObjectName`: `bean:name=testBean`. The set of interfaces that the proxy will implement is controlled by the `proxyInterfaces` property and the rules for mapping methods and properties on these interfaces to operations and attributes on the MBean are the same rules used by the `InterfaceBasedMBeanInfoAssembler`.

The `MBeanProxyFactoryBean` can create a proxy to any MBean that is accessible via an `MBeanServerConnection`. By default, the local `MBeanServer` is located and used, but you can override this and provide an `MBeanServerConnection` pointing to a remote `MBeanServer` to cater for proxies pointing to remote MBeans:

```
<bean id="clientConnector"
      class="org.springframework.jmx.support.MBeanServerConnectionFactoryBean">
  <property name="serviceUrl" value="service:jmx:rmi://remotehost:9875"/>
</bean>
```

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
  <property name="objectName" value="bean:name=testBean"/>
  <property name="proxyInterface" value="org.springframework.jmx.IJmxTestBean"/>
  <property name="server" ref="clientConnector"/>
</bean>
```

Here you can see that we create an `MBeanServerConnection` pointing to a remote machine using the `MBeanServerConnectionFactoryBean`. This `MBeanServerConnection` is then passed to the `MBeanProxyFactoryBean` via the `server` property. The proxy that is created will forward all invocations to the `MBeanServer` via this `MBeanServerConnection`.

20.7. Notifications

Spring's JMX offering includes comprehensive support for JMX notifications.

20.7.1. Registering Listeners for Notifications

Spring's JMX support makes it very easy to register any number of `NotificationListeners` with any number of MBeans (this includes MBeans exported by Spring's `MBeanExporter` and MBeans registered via some other mechanism). An example will best illustrate how simple it is to effect the registration of `NotificationListeners`. Consider the scenario where one would like to be informed (via a `Notification`) each and every time an attribute of a target MBean changes.

```
package com.example;

import javax.management.AttributeChangeNotification;
import javax.management.Notification;
import javax.management.NotificationFilter;
import javax.management.NotificationListener;

public class ConsoleLoggingNotificationListener
               implements NotificationListener, NotificationFilter {

    public void handleNotification(Notification notification, Object handback) {
        System.out.println(notification);
        System.out.println(handback);
    }

    public boolean isNotificationEnabled(Notification notification) {
        return AttributeChangeNotification.class.isAssignableFrom(notification.getClass());
    }
}
```

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListenerMappings">
        <map>
            <entry key="bean:name=testBean1">
                <bean class="com.example.ConsoleLoggingNotificationListener"/>
            </entry>
        </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>
```

```
</beans>
```

With the above configuration in place, every time a JMX `Notification` is broadcast from the target MBean (`bean:name=testBean1`), the `ConsoleLoggingNotificationListener` bean that was registered as a listener via the `notificationListenerMappings` property will be notified. The `ConsoleLoggingNotificationListener` bean can then take whatever action it deems appropriate in response to the `Notification`.

If one wants to register a single `NotificationListener` instance for all of the beans that the enclosing `MBeanExporter` is exporting, one can use the special wildcard `'*'` (sans quotes) as the key for an entry in the `notificationListenerMappings` property map; for example:

```
    <property name="notificationListenerMappings">
        <map>
            <entry key="*">
                <bean class="com.example.ConsoleLoggingNotificationListener"/>
            </entry>
        </map>
    </property>
```

If one needs to do the inverse (i.e. register a number of distinct listeners against an MBean), then one has to use the `notificationListeners` list property instead (and in preference to the `notificationListenerMappings` property). This time, instead of configuring simply a `NotificationListener` for a single MBean, one configures `NotificationListenerBean` instances... a `NotificationListenerBean` encapsulates a `NotificationListener` and the `ObjectName` (or `ObjectNames`) that it is to be registered against in an `MBeanServer`. The `NotificationListenerBean` also encapsulates a number of other properties such as a `NotificationFilter` and an arbitrary handback object that can be used in advanced JMX notification scenarios.

The configuration when using `NotificationListenerBean` instances is not wildly different to what was presented previously:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="notificationListeners">
        <list>
            <bean class="org.springframework.jmx.export.NotificationListenerBean">
                <constructor-arg>
                    <bean class="com.example.ConsoleLoggingNotificationListener"/>
                </constructor-arg>
                <property name="mappedObjectNames">
                    <list>
                        <value>bean:name=testBean1</value>
                    </list>
                </property>
            </bean>
        </list>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

The above example is equivalent to the first notification example. Lets assume then that we want to be given a

handback object every time a `Notification` is raised, and that additionally we want to filter out extraneous `Notifications` by supplying a `NotificationFilter`. (For a full discussion of just what a handback object is, and indeed what a `NotificationFilter` is, please do consult that section of the JMX specification (1.2) entitled `'The JMX Notification Model'`.)

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean1"/>
        <entry key="bean:name=testBean2" value-ref="testBean2"/>
      </map>
    </property>
    <property name="notificationListeners">
        <list>
            <bean class="org.springframework.jmx.export.NotificationListenerBean">
                <constructor-arg ref="customerNotificationListener"/>
                <property name="mappedObjectNames">
                    <list>
                        <!-- let's handle notifications from two distinct MBeans -->
                        <value>bean:name=testBean1</value>
                        <value>bean:name=testBean2</value>
                    </list>
                </property>
                <property name="handback">
                    <bean class="java.lang.String">
                        <constructor-arg value="This could be anything..."/>
                    </bean>
                </property>
                <property name="notificationFilter" ref="customerNotificationListener"/>
            </bean>
        </list>
    </property>
  </bean>

  <!-- implements both the 'NotificationListener' and 'NotificationFilter' interfaces -->
  <bean id="customerNotificationListener" class="com.example.ConsoleLoggingNotificationListener"/>

  <bean id="testBean1" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="testBean2" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="ANOTHER TEST"/>
    <property name="age" value="200"/>
  </bean>

</beans>
```

20.7.2. Publishing Notifications

Spring provides support not just for registering to receive `Notifications`, but also for publishing `Notifications`.

Note

Please note that this section is really only relevant to Spring managed beans that have been exposed as MBeans via an `MBeanExporter`; any existing, user-defined MBeans should use the standard JMX APIs for notification publication.

The key interface in Spring's JMX notification publication support is the `NotificationPublisher` interface (defined in the `org.springframework.jmx.export.notification` package). Any bean that is going to be exported as an MBean via an `MBeanExporter` instance can implement the related

`NotificationPublisherAware` interface to gain access to a `NotificationPublisher` instance. The `NotificationPublisherAware` interface simply supplies an instance of a `NotificationPublisher` to the implementing bean via a simple setter method, which the bean can then use to publish `Notifications`.

As stated in the Javadoc for the `NotificationPublisher` class, managed beans that are publishing events via the `NotificationPublisher` mechanism are *not* responsible for the state management of any notification listeners and the like ... Spring's JMX support will take care of handling all the JMX infrastructure issues. All one need do as an application developer is implement the `NotificationPublisherAware` interface and start publishing events using the supplied `NotificationPublisher` instance. Note that the `NotificationPublisher` will be set *after* the managed bean has been registered with an `MBeanServer`.

Using a `NotificationPublisher` instance is quite straightforward... one simply creates a JMX `Notification` instance (or an instance of an appropriate `Notification` subclass), populates the notification with the data pertinent to the event that is to be published, and one then invokes the `sendNotification(Notification)` on the `NotificationPublisher` instance, passing in the `Notification`.

Let's look at a simple example... in this scenario, exported instances of the `JmxTestBean` are going to publish a `NotificationEvent` every time the `add(int, int)` operation is invoked.

```
package org.springframework.jmx;

import org.springframework.jmx.export.notification.NotificationPublisherAware;
import org.springframework.jmx.export.notification.NotificationPublisher;
import javax.management.Notification;

public class JmxTestBean implements IJmxTestBean, NotificationPublisherAware {

    private String name;
    private int age;
    private boolean isSuperman;
    private NotificationPublisher publisher;

    // other getters and setters omitted for clarity

    public int add(int x, int y) {
        int answer = x + y;
        this.publisher.sendNotification(new Notification("add", this, 0));
        return answer;
    }

    public void dontExposeMe() {
        throw new RuntimeException();
    }

    public void setNotificationPublisher(NotificationPublisher notificationPublisher) {
        this.publisher = notificationPublisher;
    }
}
```

The `NotificationPublisher` interface and the machinery to get it all working is one of the nicer features of Spring's JMX support. It does however come with the price tag of coupling your classes to both Spring and JMX; as always, the advice here is to be pragmatic... if you need the functionality offered by the `NotificationPublisher` and you can accept the coupling to both Spring and JMX, then do so.

20.8. Further Resources

This section contains links to further resources about JMX.

- The JMX homepage at Sun
- The JMX specification (JSR-000003)

- The JMX Remote API specification (JSR-000160)
- The MX4J homepage (an Open Source implementation of various JMX specs)
- Getting Started with JMX - an introductory article from Sun.

Chapter 21. JCA CCI

21.1. Introduction

J2EE provides a specification to standardize access to enterprise information systems (EIS): the JCA (Java Connector Architecture). This specification is divided into several different parts:

- SPI (Service provider interfaces) that the connector provider must implement. These interfaces constitute a resource adapter which can be deployed on a J2EE application server. In such a scenario, the server manages connection pooling, transaction and security (managed mode). The application server is also responsible for managing the configuration, which is held outside the client application. A connector can be used without an application server as well; in this case, the application must configure it directly (non-managed mode).
- CCI (Common Client Interface) that an application can use to interact with the connector and thus communicate with an EIS. An API for local transaction demarcation is provided as well.

The aim of the Spring CCI support is to provide classes to access a CCI connector in typical Spring style, leveraging the Spring Framework's general resource and transaction management facilities.

Note

The client side of connectors doesn't alway use CCI. Some connectors expose their own APIs, only providing JCA resource adapter to use the system contracts of a J2EE container (connection pooling, global transactions, security). Spring does not offer special support for such connector-specific APIs.

21.2. Configuring CCI

21.2.1. Connector configuration

The base resource to use JCA CCI is the `ConnectionFactory` interface. The connector used must provide an implementation of this interface.

To use your connector, you can deploy it on your application server and fetch the `ConnectionFactory` from the server's JNDI environment (managed mode). The connector must be packaged as a RAR file (resource adapter archive) and contain a `ra.xml` file to describe its deployment characteristics. The actual name of the resource is specified when you deploy it. To access it within Spring, simply use Spring's `JndiObjectFactoryBean` to fetch the factory by its JNDI name.

Another way to use a connector is to embed it in your application (non-managed mode), not using an application server to deploy and configure it. Spring offers the possibility to configure a connector as a bean, through a provided `FactoryBean` (`LocalConnectionFactoryBean`). In this manner, you only need the connector library in the classpath (no RAR file and no `ra.xml` descriptor needed). The library must be extracted from the connector's RAR file, if necessary.

Once you have got access to your `ConnectionFactory` instance, you can inject it into your components. These components can either be coded against the plain CCI API or leverage Spring's support classes for CCI access (e.g. `CciTemplate`).

Note

When you use a connector in non-managed mode, you can't use global transactions because the resource is never enlisted / delisted in the current global transaction of the current thread. The resource is simply not aware of any global J2EE transactions that might be running.

21.2.2. `ConnectionFactory` configuration in Spring

In order to make connections to the EIS, you need to obtain a `ConnectionFactory` from the application server if you are in a managed mode, or directly from Spring if you are in a non-managed mode.

In a managed mode, you access a `ConnectionFactory` from JNDI; its properties will be configured in the application server.

```
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>
```

In non-managed mode, you must configure the `ConnectionFactory` you want to use in the configuration of Spring as a JavaBean. The `LocalConnectionFactoryBean` class offers this setup style, passing in the `ManagedConnectionFactory` implementation of your connector, exposing the application-level CCI `ConnectionFactory`.

```
<bean id="eciManagedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="tcp://localhost/"/>
  <property name="portNumber" value="2006"/>
</bean>

<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>
```

Note

You can't directly instantiate a specific `ConnectionFactory`. You need to go through the corresponding implementation of the `ManagedConnectionFactory` interface for your connector. This interface is part of the JCA SPI specification.

21.2.3. Configuring CCI connections

JCA CCI allow the developer to configure the connections to the EIS using the `ConnectionSpec` implementation of your connector. In order to configure its properties, you need to wrap the target connection factory with a dedicated adapter, `ConnectionSpecConnectionFactoryAdapter`. So, the dedicated `ConnectionSpec` can be configured with the property `connectionSpec` (as an inner bean).

This property is not mandatory because the CCI `ConnectionFactory` interface defines two different methods to obtain a CCI connection. Some of the `ConnectionSpec` properties can often be configured in the application server (in managed mode) or on the corresponding local `ManagedConnectionFactory` implementation.

```
public interface ConnectionFactory implements Serializable, Referenceable {
  ...
  Connection getConnection() throws ResourceException;
  Connection getConnection(ConnectionSpec connectionSpec) throws ResourceException;
  ...
```

```
}
```

Spring provides a `ConnectionSpecConnectionFactoryAdapter` that allows for specifying a `ConnectionSpec` instance to use for all operations on a given factory. If the adapter's `connectionSpec` property is specified, the adapter uses the `getConnection` variant without argument, else the one with the `ConnectionSpec` argument.

```
<bean id="managedConnectionFactory"
    class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>
```

21.2.4. Using a single CCI connection

If you want to use a single CCI connection, Spring provides a further `ConnectionFactory` adapter to manage this. The `SingleConnectionFactory` adapter class will open a single connection lazily and close it when this bean is destroyed at application shutdown. This class will expose special `Connection` proxies that behave accordingly, all sharing the same underlying physical connection.

```
<bean id="eciManagedConnectionFactory"
    class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TEST"/>
  <property name="connectionURL" value="tcp://localhost/"/>
  <property name="portNumber" value="2006"/>
</bean>

<bean id="targetEciConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="eciManagedConnectionFactory"/>
</bean>

<bean id="eciConnectionFactory"
    class="org.springframework.jca.cci.connection.SingleConnectionFactory">
  <property name="targetConnectionFactory" ref="targetEciConnectionFactory"/>
</bean>
```

Note

This `ConnectionFactory` adapter cannot directly be configured with a `ConnectionSpec`. Use an intermediary `ConnectionSpecConnectionFactoryAdapter` that the `SingleConnectionFactory` talks to if you require a single connection for a specific `ConnectionSpec`.

21.3. Using Spring's CCI access support

21.3.1. Record conversion

One of the aims of the JCA CCI support is to provide convenient facilities for manipulating CCI records. The developer can specify the strategy to create records and extract datas from records, for use with Spring's `CciTemplate`. The following interfaces will configure the strategy to use input and output records if you don't want to work with records directly in your application.

In order to create an input `Record`, the developer can use a dedicated implementation of the `RecordCreator` interface.

```
public interface RecordCreator {

  Record createRecord(RecordFactory recordFactory) throws ResourceException, DataAccessException;
}
```

As you can see, the `createRecord(..)` method receives a `RecordFactory` instance as parameter, which corresponds to the `RecordFactory` of the `ConnectionFactory` used. This reference can be used to create `IndexedRecord` or `MappedRecord` instances. The following sample shows how to use the `RecordCreator` interface and indexed/mapped records.

```
public class MyRecordCreator implements RecordCreator {

  public Record createRecord(RecordFactory recordFactory) throws ResourceException {
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }
}
```

An output `Record` can be used to receive data back from the EIS. Hence, a specific implementation of the `RecordExtractor` interface can be passed to Spring's `CciTemplate` for extracting data from the output `Record`.

```
public interface RecordExtractor {

  Object extractData(Record record) throws ResourceException, SQLException, DataAccessException;
}
```

The following sample shows how to use the `RecordExtractor` interface.

```
public class MyRecordExtractor implements RecordExtractor {

  public Object extractData(Record record) throws ResourceException {
    CommAreaRecord commAreaRecord = (CommAreaRecord) record;
    String str = new String(commAreaRecord.toByteArray());
    String field1 = string.substring(0,6);
    String field2 = string.substring(6,1);
    return new OutputObject(Long.parseLong(field1), field2);
  }
}
```

21.3.2. The `CciTemplate`

The `CciTemplate` is the central class of the core CCI support package (`org.springframework.jca.cci.core`). It simplifies the use of CCI since it handles the creation and release of resources. This helps to avoid common errors like forgetting to always close the connection. It cares for the lifecycle of connection and interaction objects, letting application code focus on generating input records from application data and extracting application data from output records.

The JCA CCI specification defines two distinct methods to call operations on an EIS. The CCI `Interaction` interface provides two execute method signatures:

```
public interface javax.resource.cci.Interaction {
  ...
  boolean execute(InteractionSpec spec, Record input, Record output) throws ResourceException;

  Record execute(InteractionSpec spec, Record input) throws ResourceException;
  ...
}
```

Depending on the template method called, `CciTemplate` will know which `execute` method to call on the interaction. In any case, a correctly initialized `InteractionSpec` instance is mandatory.

`CciTemplate.execute(..)` can be used in two ways:

- With direct `Record` arguments. In this case, you simply need to pass the CCI input record in, and the returned object be the corresponding CCI output record.
- With application objects, using record mapping. In this case, you need to provide corresponding `RecordCreator` and `RecordExtractor` instances.

With the first approach, the following methods of the template will be used. These methods directly correspond to those on the `Interaction` interface.

```
public class CciTemplate implements CciOperations {

  public Record execute(InteractionSpec spec, Record inputRecord)
      throws DataAccessException { ... }

  public void execute(InteractionSpec spec, Record inputRecord, Record outputRecord)
      throws DataAccessException { ... }

}
```

With the second approach, we need to specify the record creation and record extraction strategies as arguments. The interfaces used are those describe in the previous section on record conversion. The corresponding `CciTemplate` methods are the following:

```
public class CciTemplate implements CciOperations {

  public Record execute(InteractionSpec spec, RecordCreator inputCreator)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, Record inputRecord, RecordExtractor outputExtractor)
      throws DataAccessException { ... }

  public Object execute(InteractionSpec spec, RecordCreator creator, RecordExtractor extractor)
      throws DataAccessException { ... }

}
```

Unless the `outputRecordCreator` property is set on the template (see the following section), every method will call the corresponding `execute` method of the CCI `Interaction` with two parameters: `InteractionSpec` and input `Record`, receiving an output `Record` as return value.

`CciTemplate` also provides methods to create `IndexRecord` and `MappedRecord` outside a `RecordCreator` implementation, through its `createIndexRecord(..)` and `createMappedRecord(..)` methods. This can be used within DAO implementations to create `Record` instances to pass into corresponding `CciTemplate.execute(..)` methods.

```
public class CciTemplate implements CciOperations {

  public IndexedRecord createIndexedRecord(String name) throws DataAccessException { ... }

  public MappedRecord createMappedRecord(String name) throws DataAccessException { ... }

}
```

21.3.3. DAO support

Spring's CCI support provides a abstract class for DAOs, supporting injection of a `ConnectionFactory` or a `CciTemplate` instances. The name of the class is `CciDaoSupport`: It provides simple `setConnectionFactory` and `setCciTemplate` methods. Internally, this class will create a `CciTemplate` instance for a passed-in `ConnectionFactory`, exposing it to concrete data access implementations in subclasses.

```
public abstract class CciDaoSupport {

  public void setConnectionFactory(ConnectionFactory connectionFactory) { ... }
  public ConnectionFactory getConnectionFactory() { ... }

  public void setCciTemplate(CciTemplate cciTemplate) { ... }
  public CciTemplate getCciTemplate() { ... }

}
```

21.3.4. Automatic output record generation

If the connector used only supports the `Interaction.execute(..)` method with input and output records as parameters (that is, it requires the desired output record to be passed in instead of returning an appropriate output record), you can set the `outputRecordCreator` property of the `CciTemplate` to automatically generate an output record to be filled by the JCA connector when the response is received. This record will be then returned to the caller of the template.

This property simply holds an implementation of the `RecordCreator` interface, used for that purpose. The `RecordCreator` interface has already been discussed in the section entitled Section 21.3.1, “Record conversion”. The `outputRecordCreator` property must be directly specified on the `CciTemplate`. This could be done in the application code like so:

```
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());
```

Or (recommended) in the Spring configuration, if the `CciTemplate` is configured as a dedicated bean instance:

```
<bean id="eciOutputRecordCreator" class="eci.EciOutputRecordCreator"/>

<bean id="cciTemplate" class="org.springframework.jca.cci.core.CciTemplate">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
  <property name="outputRecordCreator" ref="eciOutputRecordCreator"/>
</bean>
```

Note

As the `CciTemplate` class is thread-safe, it will usually be configured as a shared instance.

21.3.5. Summary

The following table summarizes the mechanisms of the `CciTemplate` class and the corresponding methods called on the CCI `Interaction` interface:

Table 21.1. Usage of `Interaction` execute methods

CciTemplate method signature	CciTemplate outputRecordCreator property	execute method called on the CCI Interaction
Record execute(InteractionSpec, Record)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, Record)	set	boolean execute(InteractionSpec, Record, Record)
void execute(InteractionSpec, Record, Record)	not set	void execute(InteractionSpec, Record, Record)
void execute(InteractionSpec, Record, Record)	set	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, RecordCreator)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, RecordCreator)	set	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, Record, RecordExtractor)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, Record, RecordExtractor)	set	void execute(InteractionSpec, Record, Record)
Record execute(InteractionSpec, RecordCreator, RecordExtractor)	not set	Record execute(InteractionSpec, Record)
Record execute(InteractionSpec, RecordCreator, RecordExtractor)	set	void execute(InteractionSpec, Record, Record)

21.3.6. Using a CCI `Connection` and `Interaction` directly

`CciTemplate` also offers the possibility to work directly with CCI connections and interactions, in the same manner as `JdbcTemplate` and `JmsTemplate`. This is useful when you want to perform multiple operations on a CCI connection or interaction, for example.

The interface `ConnectionCallback` provides a CCI `Connection` as argument, in order to perform custom operations on it, plus the CCI `ConnectionFactory` which the `Connection` was created with. The latter can be useful for example to get an associated `RecordFactory` instance and create indexed/mapped records, for example.

```
public interface ConnectionCallback {

  Object doInConnection(Connection connection, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

The interface `InteractionCallback` provides the CCI `Interaction`, in order to perform custom operations on

it, plus the corresponding CCI `ConnectionFactory`.

```
public interface InteractionCallback {

  Object doInInteraction(Interaction interaction, ConnectionFactory connectionFactory)
      throws ResourceException, SQLException, DataAccessException;
}
```

Note

`InteractionSpec` objects can either be shared across multiple template calls or newly created inside every callback method. This is completely up to the DAO implementation.

21.3.7. Example for `CciTemplate` usage

In this section, the usage of the `CciTemplate` will be shown to acces to a CICS with ECI mode, with the IBM CICS ECI connector.

Firstly, some initializations on the CCI `InteractionSpec` must be done to specify which CICS program to access and how to interact with it.

```
ECIInteractionSpec interactionSpec = new ECIInteractionSpec();
interactionSpec.setFunctionName("MYPROG");
interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
```

Then the program can use CCI via Spring's template and specify mappings between custom objects and CCI `Records`.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ECIInteractionSpec interactionSpec = ...;

    OutputObject output = (ObjectOutput) getCciTemplate().execute(interactionSpec,
        new RecordCreator() {
          public Record createRecord(RecordFactory recordFactory) throws ResourceException {
            return new CommAreaRecord(input.toString().getBytes());
          }
        },
        new RecordExtractor() {
          public Object extractData(Record record) throws ResourceException {
            CommAreaRecord commAreaRecord = (CommAreaRecord)record;
            String str = new String(commAreaRecord.toByteArray());
            String field1 = string.substring(0,6);
            String field2 = string.substring(6,1);
            return new OutputObject(Long.parseLong(field1), field2);
          }
        });

    return output;
  }
}
```

As discussed previously, callbacks can be used to work directly on CCI connections or interactions.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(InputObject input) {
    ObjectOutput output = (ObjectOutput) getCciTemplate().execute(
        new ConnectionCallback() {
          public Object doInConnection(Connection connection, ConnectionFactory factory)
              throws ResourceException {

            // do something...
```

```
        }
      });
    }
    return output;
  }
}
```

Note

With a `ConnectionCallback`, the `Connection` used will be managed and closed by the `CciTemplate`, but any interactions created on the connection must be managed by the callback implementation.

For a more specific callback, you can implement an `InteractionCallback`. The passed-in `Interaction` will be managed and closed by the `CciTemplate` in this case.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public String getData(String input) {
    ECIInteractionSpec interactionSpec = ...;

    String output = (String) getCciTemplate().execute(interactionSpec,
        new InteractionCallback() {
          public Object doInInteraction(Interaction interaction, ConnectionFactory factory)
              throws ResourceException {
            Record input = new CommAreaRecord(inputString.getBytes());
            Record output = new CommAreaRecord();
            interaction.execute(holder.getInteractionSpec(), input, output);
            return new String(output.toByteArray());
          }
        });

    return output;
  }
}
```

For the examples above, the corresponding configuration of the involved Spring beans could look like this in non-managed mode:

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="mypackage.MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

In managed mode (that is, in a J2EE environment), the configuration could look as follows:

```
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

21.4. Modeling CCI access as operation objects

The `org.springframework.jca.cci.object` package contains support classes that allow you to access the EIS in a different style: through reusable operation objects, analogous to Spring's JDBC operation objects (see JDBC chapter). This will usually encapsulate the CCI API: an application-level input object will be passed to the operation object, so it can construct the input record and then convert the received record data to an application-level output object and return it.

Note: This approach is internally based on the `CciTemplate` class and the `RecordCreator` / `RecordExtractor` interfaces, reusing the machinery of Spring's core CCI support.

21.4.1. `MappingRecordOperation`

`MappingRecordOperation` essentially performs the same work as `CciTemplate`, but represents a specific, pre-configured operation as an object. It provides two template methods to specify how to convert an input object to a input record, and how to convert an output record to an output object (record mapping):

- `createInputRecord(..)` to specify how to convert an input object to an input `Record`
- `extractOutputData(..)` to specify how to extract an output object from an output `Record`

Here are the signatures of these methods:

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  protected abstract Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException, DataAccessException { ... }

  protected abstract Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException, DataAccessException { ... }
  ...
}
```

Thereafter, in order to execute an EIS operation, you need to use a single execute method, passing in an application-level input object and receiving an application-level output object as result:

```
public abstract class MappingRecordOperation extends EisOperation {
  ...
  public Object execute(Object inputObject) throws DataAccessException {
  ...
}
```

As you can see, contrary to the `CciTemplate` class, this `execute(..)` method does not have an `InteractionSpec` as argument. Instead, the `InteractionSpec` is global to the operation. The following constructor must be used to instantiate an operation object with a specific `InteractionSpec`:

```
InteractionSpec spec = ...;
MyMappingRecordOperation eisOperation = new MyMappingRecordOperation(getConnectionFactory(), spec);
...
```

21.4.2. `MappingCommAreaOperation`

Some connectors use records based on a COMMAREA which represents an array of bytes containing parameters to send to the EIS and data returned by it. Spring provides a special operation class for working

directly on COMMAREA rather than on records. The `MappingCommAreaOperation` class extends the `MappingRecordOperation` class to provide such special COMMAREA support. It implicitly uses the `CommAreaRecord` class as input and output record type, and provides two new methods to convert an input object into an input COMMAREA and the output COMMAREA into an output object.

```
public abstract class MappingCommAreaOperation extends MappingRecordOperation {
  ...
  protected abstract byte[] objectToBytes(Object inObject)
      throws IOException, DataAccessException;

  protected abstract Object bytesToObject(byte[] bytes)
      throws IOException, DataAccessException;
  ...
}
```

21.4.3. Automatic output record generation

As every `MappingRecordOperation` subclass is based on CciTemplate internally, the same way to automatically generate output records as with `CciTemplate` is available. Every operation object provides a corresponding `setOutputRecordCreator(..)` method. For further information, see the section entitled Section 21.3.4, “Automatic output record generation”.

21.4.4. Summary

The operation object approach uses records in the same manner as the `CciTemplate` class.

Table 21.2. Usage of Interaction execute methods

`MappingRecordOperation` method signature	**`MappingRecordOperation` `outputRecordCreator` property**	**execute method called on the CCI `Interaction`**
Object execute(Object)	not set	Record execute(InteractionSpec, Record)
Object execute(Object)	set	boolean execute(InteractionSpec, Record, Record)

21.4.5. Example for `MappingRecordOperation` usage

In this section, the usage of the `MappingRecordOperation` will be shown to access a database with the Blackbox CCI connector.

Note

The original version of this connector is provided by the J2EE SDK (version 1.3), available from Sun.

Firstly, some initializations on the CCI `InteractionSpec` must be done to specify which SQL request to execute. In this sample, we directly define the way to convert the parameters of the request to a CCI record and the way to convert the CCI result record to an instance of the `Person` class.

```
public class PersonMappingOperation extends MappingRecordOperation {

```

```
  public PersonMappingOperation(ConnectionFactory connectionFactory) {
    setConnectionFactory(connectionFactory);
    CciInteractionSpec interactionSpec = new CciConnectionSpec();
    interactionSpec.setSql("select * from person where person_id=?");
    setInteractionSpec(interactionSpec);
  }

  protected Record createInputRecord(RecordFactory recordFactory, Object inputObject)
      throws ResourceException {
    Integer id = (Integer) inputObject;
    IndexedRecord input = recordFactory.createIndexedRecord("input");
    input.add(new Integer(id));
    return input;
  }

  protected Object extractOutputData(Record outputRecord)
      throws ResourceException, SQLException {
    ResultSet rs = (ResultSet) outputRecord;
    Person person = null;
    if (rs.next()) {
      Person person = new Person();
      person.setId(rs.getInt("person_id"));
      person.setLastName(rs.getString("person_last_name"));
      person.setFirstName(rs.getString("person_first_name"));
    }
    return person;
  }
}
```

Then the application can execute the operation object, with the person identifier as argument. Note that operation object could be set up as shared instance, as it is thread-safe.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public Person getPerson(int id) {
    PersonMappingOperation query = new PersonMappingOperation(getConnectionFactory());
    Person person = (Person) query.execute(new Integer(id));
    return person;
  }
}
```

The corresponding configuration of Spring beans could look as follows in non-managed mode:

```
<bean id="managedConnectionFactory"
    class="com.sun.connector.cciblackbox.CciLocalTxManagedConnectionFactory">
  <property name="connectionURL" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="driverName" value="org.hsqldb.jdbcDriver"/>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

In managed mode (that is, in a J2EE environment), the configuration could look as follows:

```
<bean id="targetConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/blackbox"/>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory" ref="targetConnectionFactory"/>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user" value="sa"/>
      <property name="password" value=""/>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

21.4.6. Example for `MappingCommAreaOperation` usage

In this section, the usage of the `MappingCommAreaOperation` will be shown: accessing a CICS with ECI mode with the IBM CICS ECI connector.

Firstly, the CCI `InteractionSpec` needs to be initialized to specify which CICS program to access and how to interact with it.

```
public abstract class EciMappingOperation extends MappingCommAreaOperation {

  public EciMappingOperation(ConnectionFactory connectionFactory, String programName) {
    setConnectionFactory(connectionFactory);
    ECIInteractionSpec interactionSpec = new ECIInteractionSpec(),
    interactionSpec.setFunctionName(programName);
    interactionSpec.setInteractionVerb(ECIInteractionSpec.SYNC_SEND_RECEIVE);
    interactionSpec.setCommareaLength(30);
    setInteractionSpec(interactionSpec);
    setOutputRecordCreator(new EciOutputRecordCreator());
  }

  private static class EciOutputRecordCreator implements RecordCreator {
    public Record createRecord(RecordFactory recordFactory) throws ResourceException {
      return new CommAreaRecord();
    }
  }
}
```

The abstract `EciMappingOperation` class can then be subclassed to specify mappings between custom objects and `Records`.

```
public class MyDaoImpl extends CciDaoSupport implements MyDao {

  public OutputObject getData(Integer id) {
    EciMappingOperation query = new EciMappingOperation(getConnectionFactory(), "MYPROG") {
      protected abstract byte[] objectToBytes(Object inObject) throws IOException {
        Integer id = (Integer) inObject;
        return String.valueOf(id);
      }
      protected abstract Object bytesToObject(byte[] bytes) throws IOException;
        String str = new String(bytes);
        String field1 = str.substring(0,6);
        String field2 = str.substring(6,1);
        String field3 = str.substring(7,1);
        return new OutputObject(field1, field2, field3);
      }
    });

    return (OutputObject) query.execute(new Integer(id));
  }
}
```

The corresponding configuration of Spring beans could look as follows in non-managed mode:

```
<bean id="managedConnectionFactory" class="com.ibm.connector2.cics.ECIManagedConnectionFactory">
  <property name="serverName" value="TXSERIES"/>
  <property name="connectionURL" value="local:"/>
  <property name="userName" value="CICSUSER"/>
  <property name="password" value="CICS"/>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory" ref="managedConnectionFactory"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

In managed mode (that is, in a J2EE environment), the configuration could look as follows:

```
<bean id="connectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

21.5. Transactions

JCA specifies several levels of transaction support for resource adapters. The kind of transactions that your resource adapter supports is specified in its `ra.xml` file. There are essentially three options: none (for example with CICS EPI connector), local transactions (for example with a CICS ECI connector), global transactions (for example with an IMS connector).

```
<connector>

  <resourceadapter>

    <!-- <transaction-support>NoTransaction</transaction-support> -->
    <!-- <transaction-support>LocalTransaction</transaction-support> -->
    <transaction-support>XATransaction</transaction-support>

  <resourceadapter>

<connector>
```

For global transactions, you can use Spring's generic transaction infrastructure to demarcate transactions, with `JtaTransactionManager` as backend (delegating to the J2EE server's distributed transaction coordinator underneath).

For local transactions on a single CCI `ConnectionFactory`, Spring provides a specific transaction management strategy for CCI, analogous to the `DataSourceTransactionManager` for JDBC. The CCI API defines a local transaction object and corresponding local transaction demarcation methods. Spring's `CciLocalTransactionManager` executes such local CCI transactions, fully compliant with Spring's generic `PlatformTransactionManager` abstraction.

```
<bean id="eciConnectionFactory" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName" value="eis/cicseci"/>
</bean>

<bean id="eciTransactionManager"
    class="org.springframework.jca.cci.connection.CciLocalTransactionManager">
  <property name="connectionFactory" ref="eciConnectionFactory"/>
```

```
</bean>
```

Both transaction strategies can be used with any of Spring's transaction demarcation facilities, be it declarative or programmatic. This is a consequence of Spring's generic `PlatformTransactionManager` abstraction, which decouples transaction demarcation from the actual execution strategy. Simply switch between `JtaTransactionManager` and `CciLocalTransactionManager` as needed, keeping your transaction demarcation as-is.

For more information on Spring's transaction facilities, see the chapter entitled Chapter 9, *Transaction management*.

Chapter 22. Email

22.1. Introduction

Library dependencies

The following additional jars to be on the classpath of your application in order to be able to use the Spring Framework's email library.

- The JavaMail `mail.jar` library
- The JAF `activation.jar` library

All of these libraries are available in the Spring-with-dependencies distribution of the Spring Framework (in addition to also being freely available on the web).

The Spring Framework provides a helpful utility library for sending email that shields the user from the specifics of the underlying mailing system and is responsible for low level resource handling on behalf of the client.

The `org.springframework.mail` package is the root level package for the Spring Framework's email support. The central interface for sending emails is the `MailSender` interface; a simple value object encapsulating the properties of a simple mail such as *from* and *to* (plus many others) is the `SimpleMailMessage` class. This package also contains a hierarchy of checked exceptions which provide a higher level of abstraction over the lower level mail system exceptions with the root exception being `MailException`. Please refer to the Javadocs for more information on the rich mail exception hierarchy.

The `org.springframework.mail.javamail.JavaMailSender` interface adds specialized *JavaMail* features such as MIME message support to the `MailSender` interface (from which it inherits). `JavaMailSender` also provides a callback interface for preparation of JavaMail MIME messages, called `org.springframework.mail.javamail.MimeMessagePreparator`

22.2. Usage

Let's assume there is a business interface called `OrderManager`:

```
public interface OrderManager {

    void placeOrder(Order order);
}
```

Let us also assume that there is a requirement stating that an email message with an order number needs to be generated and sent to a customer placing the relevant order.

22.2.1. Basic `MailSender` and `SimpleMailMessage` usage

```
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;
```

```
public class SimpleOrderManager implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage templateMessage;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setTemplateMessage(SimpleMailMessage templateMessage) {
        this.templateMessage = templateMessage;
    }

    public void placeOrder(Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        // Create a thread safe "copy" of the template message and customize it
        SimpleMailMessage msg = new SimpleMailMessage(this.templateMessage);
        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{
            this.mailSender.send(msg);
        }
        catch(MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }
}
```

Find below the bean definitions for the above code:

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host" value="mail.mycompany.com"/>
</bean>

<!-- this is a template message that we can pre-load with default state -->
<bean id="templateMessage" class="org.springframework.mail.SimpleMailMessage">
  <property name="from" value="customerservice@mycompany.com"/>
  <property name="subject" value="Your order"/>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.SimpleOrderManager">
  <property name="mailSender" ref="mailSender"/>
  <property name="templateMessage" ref="templateMessage"/>
</bean>
```

22.2.2. Using the `JavaMailSender` and the `MimeMessagePreparator`

Here is another implementation of `OrderManager` using the `MimeMessagePreparator` callback interface. Please note in this case that the `mailSender` property is of type `JavaMailSender` so that we are able to use the JavaMail `MimeMessage` class:

```
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 SimpleOrderManager implements OrderManager {
```

```
    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void placeOrder(final Order order) {

        // Do the business calculations...

        // Call the collaborators to persist the order...

        MimeMessagePreparator preparator = new MimeMessagePreparator() {

            public void prepare(MimeMessage mimeMessage) throws Exception {

                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 {
            this.mailSender.send(preparator);
        }
        catch (MailException ex) {
            // simply log it and go on...
            System.err.println(ex.getMessage());
        }
    }
}
```

Note

The mail code is a crosscutting concern and could well be a candidate for refactoring into a custom Spring AOP aspect, which then could be executed at appropriate joinpoints on the `OrderManager` target.

The Spring Framework's mail support ships with two `MailSender` implementations. The standard JavaMail implementation and the implementation on top of Jason Hunter's `MailMessage` class that is included in [the `com.oreilly.servlet` package](). Please refer to the relevant Javadocs for more information.

22.3. Using the JavaMail `MimeMessageHelper`

A class that comes in pretty handy when dealing with JavaMail messages is the `org.springframework.mail.javamail.MimeMessageHelper` class, which shields you from having to use the verbose JavaMail API. Using the `MimeMessageHelper` it is pretty easy to create a `MimeMessage`:

```
// of course you would use DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("test@host.com");
helper.setText("Thank you for ordering!");

sender.send(message);
```

22.3.1. Sending attachments and inline resources

Multipart email messages allow for both attachments and inline resources. Examples of inline resources would be be images or a stylesheet you want to use in your message, but that you don't want displayed as an attachment.

22.3.1.1. Attachments

The following example shows you how to use the `MimeMessageHelper` to send an email along with a single JPEG image attachment.

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

helper.setText("Check out this image!");

// let's attach the infamous windows Sample file (this time copied to c:/)
FileSystemResource file = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addAttachment("CoolImage.jpg", file);

sender.send(message);
```

22.3.1.2. Inline resources

The following example shows you how to use the `MimeMessageHelper` to send an email along with an inline image.

```
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMessage();

// use the true flag to indicate you need a multipart message
MimeMessageHelper helper = new MimeMessageHelper(message, true);
helper.setTo("test@host.com");

// use the true flag to indicate the text included is HTML
helper.setText("<html><body><img src='cid:identifier1234'></body></html>", true);

// let's include the infamous windows Sample file (this time copied to c:/)
FileSystemResource res = new FileSystemResource(new File("c:/Sample.jpg"));
helper.addInline("identifier1234", res);

sender.send(message);
```

Warning

Inline resources are added to the mime message using the specified `Content-ID` (`identifier1234` in the above example). The order in which you are adding the text and the resource are **very** important. Be sure to *first add the text* and after that the resources. If you are doing it the other way around, it won't work!

22.3.2. Creating email content using a templating library

The code in the previous examples explicitly has been creating the content of the email message, using methods calls such as `message.setText(..)`. This is fine for simple cases, and it is okay in the context of the aforementioned examples, where the intent was to show you the very basics of the API.

In your typical enterprise application though, you are not going to create the content of your emails using the above approach for a number of reasons.

- Creating HTML-based email content in Java code is tedious and error prone
- There is no clear separation between display logic and business logic
- Changing the display structure of the email content requires writing Java code, recompiling, redeploying...

Typically the approach taken to address these issues is to use a template library such as FreeMarker or Velocity to define the display structure of email content. This leaves your code tasked only with creating the data that is to be rendered in the email template and sending the email. It is definitely a best practice for when the content of your emails becomes even moderately complex, and with the Spring Framework's support classes for FreeMarker and Velocity becomes quite easy to do. Find below an example of using the Velocity template library to create email content.

22.3.2.1. A Velocity-based example

To use Velocity to create your email template(s), you will need to have the Velocity libraries available on your classpath. You will also need to create one or more Velocity templates for the email content that your application needs. Find below the Velocity template that this example will be using... as you can see it is HTML-based, and since it is plain text it can be created using your favorite HTML editor without recourse to having to know Java.

```
# in the com/foo/package
<html>
<body>
<h3>Hi ${user.userName}, welcome to the Chipping Sodbury On-the-Hill message boards!</h3>

<div>
   Your email address is <a href="mailto:${user.emailAddress}">${user.emailAddress}</a>.
</div>
</body>

</html>
```

Find below some simple code and Spring XML configuration that makes use of the above Velocity template to create email content and send email(s).

```
package com.foo;

import org.apache.velocity.app.VelocityEngine;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessageHelper;
import org.springframework.mail.javamail.MimeMessagePreparator;
import org.springframework.ui.velocity.VelocityEngineUtils;

import javax.mail.internet.MimeMessage;
import java.util.HashMap;
import java.util.Map;

public class SimpleRegistrationService implements RegistrationService {

   private JavaMailSender mailSender;
   private VelocityEngine velocityEngine;

   public void setMailSender(JavaMailSender mailSender) {
```

```
      this.mailSender = mailSender;
   }

   public void setVelocityEngine(VelocityEngine velocityEngine) {
      this.velocityEngine = velocityEngine;
   }

   public void register(User user) {

      // Do the registration logic...

      sendConfirmationEmail(user);
   }

   private void sendConfirmationEmail(final User user) {
      MimeMessagePreparator preparator = new MimeMessagePreparator() {
         public void prepare(MimeMessage mimeMessage) throws Exception {
            MimeMessageHelper message = new MimeMessageHelper(mimeMessage);
            message.setTo(user.getEmailAddress());
            message.setFrom("webmaster@csonth.gov.uk"); // could be parameterized...
            Map model = new HashMap();
            model.put("user", user);
            String text = VelocityEngineUtils.mergeTemplateIntoString(
               velocityEngine, "com/dns/registration-confirmation.vm", model);
            message.setText(text, true);
         }
      };
      this.mailSender.send(preparator);
   }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

   <bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
      <property name="host" value="mail.csonth.gov.uk"/>
   </bean>

   <bean id="registrationService" class="com.foo.SimpleRegistrationService">
      <property name="mailSender" ref="mailSender"/>
      <property name="velocityEngine" ref="velocityEngine"/>
   </bean>

   <bean id="velocityEngine" class="org.springframework.ui.velocity.VelocityEngineFactoryBean">
      <property name="velocityProperties">
         <value>
            resource.loader=class
            class.resource.loader.class=org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
         </value>
      </property>
   </bean>

</beans>
```

Chapter 23. Scheduling and Thread Pooling

23.1. Introduction

The Spring Framework features integration classes for scheduling support. Currently, Spring supports the `Timer`, part of the JDK since 1.3, and the Quartz Scheduler (http://www.opensymphony.com/quartz/). Both schedulers are set up using a `FactoryBean` with optional references to `Timer` or `Trigger` instances, respectively. Furthermore, a convenience class for both the Quartz Scheduler and the `Timer` is available that allows you to invoke a method of an existing target object (analogous to the normal `MethodInvokingFactoryBean` operation). Spring also features classes for thread pooling that abstract away differences between Java 1.3, 1.4, 5 and JEE environments.

23.2. Using the OpenSymphony Quartz Scheduler

Quartz uses `Trigger`, `Job` and `JobDetail` objects to realize scheduling of all kinds of jobs. For the basic concepts behind Quartz, have a look at http://www.opensymphony.com/quartz. For convenience purposes, Spring offers a couple of classes that simplify the usage of Quartz within Spring-based applications.

23.2.1. Using the JobDetailBean

`JobDetail` objects contain all information needed to run a job. The Spring Framework provides a `JobDetailBean` that makes the `JobDetail` more of an actual JavaBean with sensible defaults. Let's have a look at an example:

```
<bean name="exampleJob" class="org.springframework.scheduling.quartz.JobDetailBean">
  <property name="jobClass" value="example.ExampleJob" />
  <property name="jobDataAsMap">
    <map>
      <entry key="timeout" value="5" />
    </map>
  </property>
</bean>
```

The job detail bean has all information it needs to run the job (`ExampleJob`). The timeout is specified in the job data map. The job data map is available through the `JobExecutionContext` (passed to you at execution time), but the `JobDetailBean` also maps the properties from the job data map to properties of the actual job. So in this case, if the `ExampleJob` contains a property named `timeout`, the `JobDetailBean` will automatically apply it:

```
package example;

public class ExampleJob extends QuartzJobBean {

  private int timeout;

  /**
   * Setter called after the ExampleJob is instantiated
   * with the value from the JobDetailBean (5)
   */
  public void setTimeout(int timeout) {
    this.timeout = timeout;
  }

  protected void executeInternal(JobExecutionContext ctx) throws JobExecutionException {
      // do the actual work
  }
}
```

All additional settings from the job detail bean are of course available to you as well.

Note: Using the `name` and `group` properties, you can modify the name and the group of the job, respectively. By default, the name of the job matches the bean name of the job detail bean (in the example above, this is `exampleJob`).

23.2.2. Using the `MethodInvokingJobDetailFactoryBean`

Often you just need to invoke a method on a specific object. Using the `MethodInvokingJobDetailFactoryBean` you can do exactly this:

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
</bean>
```

The above example will result in the `doIt` method being called on the `exampleBusinessObject` method (see below):

```
public class ExampleBusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

```
<bean id="exampleBusinessObject" class="examples.ExampleBusinessObject"/>
```

Using the `MethodInvokingJobDetailFactoryBean`, you don't need to create one-line jobs that just invoke a method, and you only need to create the actual business object and wire up the detail object.

By default, Quartz Jobs are stateless, resulting in the possibility of jobs interfering with each other. If you specify two triggers for the same `JobDetail`, it might be possible that before the first job has finished, the second one will start. If `JobDetail` classes implement the `Stateful` interface, this won't happen. The second job will not start before the first one has finished. To make jobs resulting from the `MethodInvokingJobDetailFactoryBean` non-concurrent, set the `concurrent` flag to `false`.

```
<bean id="jobDetail" class="org.springframework.scheduling.quartz.MethodInvokingJobDetailFactoryBean">
  <property name="targetObject" ref="exampleBusinessObject" />
  <property name="targetMethod" value="doIt" />
  <property name="concurrent" value="false" />
</bean>
```

Note

By default, jobs will run in a concurrent fashion.

23.2.3. Wiring up jobs using triggers and the `SchedulerFactoryBean`

We've created job details and jobs. We've also reviewed the convenience bean that allows to you invoke a method on a specific object. Of course, we still need to schedule the jobs themselves. This is done using

triggers and a `SchedulerFactoryBean`. Several triggers are available within Quartz. Spring offers two subclassed triggers with convenient defaults: `CronTriggerBean` and `SimpleTriggerBean`.

Triggers need to be scheduled. Spring offers a `SchedulerFactoryBean` that exposes triggers to be set as properties. `SchedulerFactoryBean` schedules the actual jobs with those triggers.

Find below a couple of examples:

```
<bean id="simpleTrigger" class="org.springframework.scheduling.quartz.SimpleTriggerBean">
    <!-- see the example of method invoking job above -->
    <property name="jobDetail" ref="jobDetail" />
    <!-- 10 seconds -->
    <property name="startDelay" value="10000" />
    <!-- repeat every 50 seconds -->
    <property name="repeatInterval" value="50000" />
</bean>

<bean id="cronTrigger" class="org.springframework.scheduling.quartz.CronTriggerBean">
    <property name="jobDetail" ref="exampleJob" />
    <!-- run every morning at 6 AM -->
    <property name="cronExpression" value="0 0 6 * * ?" />
</bean>
```

Now we've set up two triggers, one running every 50 seconds with a starting delay of 10 seconds and one every morning at 6 AM. To finalize everything, we need to set up the `SchedulerFactoryBean`:

```
<bean class="org.springframework.scheduling.quartz.SchedulerFactoryBean">
    <property name="triggers">
        <list>
            <ref bean="cronTrigger" />
            <ref bean="simpleTrigger" />
        </list>
    </property>
</bean>
```

More properties are available for the `SchedulerFactoryBean` for you to set, such as the calendars used by the job details, properties to customize Quartz with, etc. Have a look at the SchedulerFactoryBean Javadoc for more information.

23.3. Using JDK Timer support

The other way to schedule jobs in Spring is to use JDK `Timer` objects. You can create custom timers or use the timer that invokes methods. Wiring timers is done using the `TimerFactoryBean`.

23.3.1. Creating custom timers

Using the `TimerTask` you can create customer timer tasks, similar to Quartz jobs:

```
public class CheckEmailAddresses extends TimerTask {

  private List emailAddresses;

  public void setEmailAddresses(List emailAddresses) {
    this.emailAddresses = emailAddresses;
  }

  public void run() {
    // iterate over all email addresses and archive them
  }
}
```

Wiring it up is simple:

```
<bean id="checkEmail" class="examples.CheckEmailAddress">
    <property name="emailAddresses">
        <list>
            <value>test@springframework.org</value>
            <value>foo@bar.com</value>
            <value>john@doe.net</value>
        </list>
    </property>
</bean>

<bean id="scheduledTask" class="org.springframework.scheduling.timer.ScheduledTimerTask">
    <!-- wait 10 seconds before starting repeated execution -->
    <property name="delay" value="10000" />
    <!-- run every 50 seconds -->
    <property name="period" value="50000" />
    <property name="timerTask" ref="checkEmail" />
</bean>
```

Note that letting the task only run once can be done by changing the `period` property to 0 (or a negative value).

23.3.2. Using the `MethodInvokingTimerTaskFactoryBean`

Similar to the Quartz support, the `Timer` support also features a component that allows you to periodically invoke a method:

```
<bean id="doIt" class="org.springframework.scheduling.timer.MethodInvokingTimerTaskFactoryBean">
    <property name="targetObject" ref="exampleBusinessObject" />
    <property name="targetMethod" value="doIt" />
</bean>
```

The above example will result in the `doIt` method being called on the `exampleBusinessObject` (see below):

```
public class BusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

Changing the `timerTask` reference of the `ScheduledTimerTask` example to the bean `doIt` will result in the `doIt` method being executed on a fixed schedule.

23.3.3. Wrapping up: setting up the tasks using the `TimerFactoryBean`

The `TimerFactoryBean` is similar to the Quartz `SchedulerFactoryBean` in that it serves the same purpose: setting up the actual scheduling. The `TimerFactoryBean` sets up an actual `Timer` and schedules the tasks it has references to. You can specify whether or not daemon threads should be used.

```
<bean id="timerFactory" class="org.springframework.scheduling.timer.TimerFactoryBean">
    <property name="scheduledTimerTasks">
        <list>
            <!-- see the example above -->
            <ref bean="scheduledTask" />
        </list>
    </property>
</bean>
```

23.4. The Spring `TaskExecutor` abstraction

23.4. The Spring `TaskExecutor` abstraction

Spring 2.0 introduces a new abstraction for dealing with Executors. Executors are the Java 5 name for the concept of thread pools. The odd naming is due to the fact that there is no guarantee that the underlying implementation is actually a pool. In fact, in many cases, the executor is single-threaded. Spring's abstraction helps bring thread pooling to Java 1.3 and 1.4 environments as well as hide implementation details between 1.3, 1.4, 5, and Java EE environments.

Spring's `TaskExecutor` interface is identical to the `java.util.concurrent.Executor` interface. In fact, its primary reason for existence is to abstract away the need for Java 5 when using thread pools. The interface has a single method `execute(Runnable task)` that accepts a task for execution based on the semantics and configuration of the thread pool.

The `TaskExecutor` was originally created to give other Spring components an abstraction for thread pooling where needed. Components such as the `ApplicationEventMulticaster`, JMS's `AbstractMessageListenerContainer`, and Quartz integration all use the `TaskExecutor` abstraction to pool threads. However, if your beans need thread pooling behavior, it is possible to use this abstraction for your own needs.

23.4.1. `TaskExecutor` types

There are a number of pre-built implementations of `TaskExecutor` included with the Spring distribution. In all likelihood, you shouldn't ever need to implement your own.

- `SimpleAsyncTaskExecutor`

  This implementation does not reuse any threads, rather it starts up a new thread for each invocation. However, it does support a concurrency limit which will block any invocations that are over the limit until a slot has been freed up. If you're looking for true pooling, keep scrolling further down the page.

- `SyncTaskExecutor`

  This implementation doesn't execute invocations asynchronously. Instead, each invocation takes place in the calling thread. It is primarily used in situations where mutlithreading isn't necessary such as simple test cases.

- `ConcurrentTaskExecutor`

  This implementation is a wrapper for a Java 5 `java.util.concurrent.Executor`. There is an alternative, `ThreadPoolTaskExecutor`, that exposes the `Executor` configuration parameters as bean properties. It is rare to need to use the `ConcurrentTaskExecutor` but if the `ThreadPoolTaskExecutor` isn't robust enough for your needs, the `ConcurrentTaskExecutor` is an alternative.

- `SimpleThreadPoolTaskExecutor`

  This implementation is actually a subclass of Quartz's `SimpleThreadPool` which listens to Spring's lifecycle callbacks. This is typically used when you have a threadpool that may need to be shared by both Quartz and non-Quartz components.

- `ThreadPoolTaskExecutor`

> It is not possible to use any backport or alternate versions of the `java.util.concurrent` package with this implementation. Both Doug Lea's and Dawid Kurzyniec's implementations use different package structures which will prevent them from working correctly.

This implementation can only be used in a Java 5 environment but is also the most commonly used one in that environment. It exposes bean properties for configuring a `java.util.concurrent.ThreadPoolExecutor` and wraps it in a `TaskExecutor`. If you need something advanced such as a `ScheduledThreadPoolExecutor`, it is recommended that you use a `ConcurrentTaskExecutor` instead.

- `TimerTaskExecutor`

This implementation uses a single `TimerTask` as its backing implementation. It's different from the `SyncTaskExecutor` in that the method invocations are executed in a separate thread, although they are synchronous in that thread.

- `WorkManagerTaskExecutor`

> CommonJ is a set of specifications jointly developed between BEA and IBM. These specifications are not Java EE standards, but are standard across BEA's and IBM's Application Server implementations.

This implementation uses the CommonJ WorkManager as its backing implementation and is the central convenience class for setting up a CommonJ WorkManager reference in a Spring context. Similar to the `SimpleThreadPoolTaskExecutor`, this class implements the WorkManager interface and therefore can be used directly as a WorkManager as well.

23.4.2. Using a `TaskExecutor`

Spring's `TaskExecutor` implementations are used as simple JavaBeans. In the example below, we define a bean that uses the `ThreadPoolTaskExecutor` to asynchronously print out a set of messages.

```
import org.springframework.core.task.TaskExecutor;

public class TaskExecutorExample {

  private class MessagePrinterTask implements Runnable {

    private String message;

    public MessagePrinterTask(String message) {
      this.message = message;
    }

    public void run() {
      System.out.println(message);
    }

  }

  private TaskExecutor taskExecutor;

  public TaskExecutorExample(TaskExecutor taskExecutor) {
    this.taskExecutor = taskExecutor;
  }
```

```
  public void printMessages() {
    for(int i = 0; i < 25; i++) {
      taskExecutor.execute(new MessagePrinterTask("Message" + i));
    }
  }
}
```

As you can see, rather than retrieving a thread from the pool and executing yourself, you add your `Runnable` to the queue and the `TaskExecutor` uses it's internal rules to decide when the task gets executed.

To configure the rules that the `TaskExecutor` will use, simple bean properties have been exposed.

```
<bean id="taskExecutor" class="org.springframework.scheduling.concurrent.ThreadPoolTaskExecutor">
  <property name="corePoolSize" value="5" />
  <property name="maxPoolSize" value="10" />
  <property name="queueCapacity" value="25" />
</bean>

<bean id="taskExecutorExample" class="TaskExecutorExample">
  <constructor-arg ref="taskExecutor" />
</bean>
```

Chapter 24. Dynamic language support

24.1. Introduction

Spring 2.0 introduces comprehensive support for using classes and objects that have been defined using a dynamic language (such as JRuby) with Spring.

> **Why only these languages?**
>
> The supported languages were chosen because a) the languages have a lot of traction in the Java enterprise community, b) no requests were made for other languages within the Spring 2.0 development timeframe, and c) the Spring developers were most familiar with them.
>
> There is nothing stopping the inclusion of further languages though. If you want to see support for <*insert your favourite dynamic language here*>, you can always raise an issue on Spring's JIRA page (or implement such support yourself).

This support allows you to write any number of classes in a supported dynamic language, and have the Spring container transparently instantiate, configure and dependency inject the resulting objects.

The dynamic languages currently supported are:

- JRuby
- Groovy
- BeanShell

Fully working examples of where this dynamic language support can be immediately useful are described in the section entitled Section 24.4, “Scenarios”.

Please note that the dynamic language support detailed in this chapter is *only* available in Spring versions 2.0 and above. Currently there are *no* plans to backport the dynamic language support to previous versions of Spring (most notably the 1.2.x line).

24.2. A first example

This bulk of this chapter is concerned with describing the dynamic language support in detail. Before diving into all of the ins and outs of the dynamic language support, let's look at a quick example of a bean defined in a dynamic language.

The dynamic language for this first bean is Groovy (the basis of this example was taken from the Spring test suite, so if you want to see equivalent examples in any of the other supported languages, take a look in the source code).

Let's look at the `Messenger` interface that the Groovy bean is going to be implementing. Note that this interface is defined in plain Java. Dependent objects that are injected with a reference to the `Messenger` won't know that the underlying implementation is a Groovy script.

```
package org.springframework.scripting;
```

```
public interface Messenger {

    String getMessage();
}
```

Here is the definition of a class that has a dependency on the `Messenger` interface.

```
package org.springframework.scripting;

public class DefaultBookingService implements BookingService {

    private Messenger messenger;

    public void setMessenger(Messenger messenger) {
        this.messenger = messenger;
    }

    public void processBooking() {
        // use the injected Messenger object...
    }
}
```

Here is an implementation of the `Messenger` interface in Groovy.

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

// import the Messenger interface (written in Java) that is to be implemented
import org.springframework.scripting.Messenger

// define the implementation in Groovy
class GroovyMessenger implements Messenger {

    String message
}
```

Finally, here are the bean definitions that will effect the injection of the Groovy-defined `Messenger` implementation into an instance of the `DefaultBookingService` class.

Note

To use the custom dynamic language tags to define dynamic-language-backed beans, you need to have the XML Schema preamble at the top of your Spring XML configuration file. You also need to be using a Spring `ApplicationContext` implementation as your IoC container. Using the dynamic-language-backed beans with a plain `BeanFactory` implementation is supported, but you have to manage the plumbing of the Spring internals to do so.

For more information on schema-based configuration, see Appendix A, *XML Schema-based configuration*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

    <!-- this is the bean definition for the Groovy-backed Messenger implementation -->
    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <!-- an otherwise normal bean that will be injected by the Groovy-backed Messenger -->
    <bean id="bookingService" class="x.y.DefaultBookingService">
```

```
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

The `bookingService` bean (a `DefaultBookingService`) can now use its private `messenger` member variable as normal because the `Messenger` instance that was injected into it *is* a `Messenger` instance. There is nothing special going on here, just plain Java and plain Groovy.

Hopefully the above XML snippet is self-explanatory, but don't worry unduly if it isn't. Keep reading for the in-depth detail on the whys and wherefores of the above configuration.

24.3. Defining beans that are backed by dynamic languages

This section describes exactly how you define Spring managed beans in any of the supported dynamic languages.

Please note that this chapter does not attempt to explain the syntax and idioms of the supported dynamic languages. For example, if you want to use Groovy to write certain of the classes in your application, then the assumption is that you already know Groovy. If you need further details about the dynamic languages themselves, please consult the section entitled Section 24.6, “Further Resources” at the end of this chapter.

24.3.1. Common concepts

The steps involved in using dynamic-language-backed beans are as follows:

1. Write the test for the dynamic language source code (naturally)
2. *Then* write the dynamic language source code itself :)
3. Define your dynamic-language-backed beans using the appropriate `<lang:language/>` element in the XML configuration (you can of course define such beans programmatically using the Spring API - although you will have to consult the source code for directions on how to do this as this type of advanced configuration is not covered in this chapter). Note this is an iterative step. You will need at least one bean definition per dynamic language source file (although the same dynamic language source file can of course be referenced by multiple bean definitions).

The first two steps (testing and writing your dynamic language source files) are beyond the scope of this chapter. Refer to the language specification and / or reference manual for your chosen dynamic language and crack on with developing your dynamic language source files. You *will* first want to read the rest of this chapter though, as Spring's dynamic language support does make some (small) assumptions about the contents of your dynamic language source files.

24.3.1.1. The `<lang:language/>` element

XML Schema

All of the configuration examples in this chapter make use of the new XML Schema support that was added in Spring 2.0.

It is possible to forego the use of XML Schema and stick with the old-style DTD based validation of your

Spring XML files, but then you lose out on the convenience offered by the `<lang:language/>` element. See the Spring test suite for examples of the older style configuration that doesn't require XML Schema-based validation (it is quite verbose and doesn't hide any of the underlying Spring implementation from you).

The final step involves defining dynamic-language-backed bean definitions, one for each bean that you want to configure (this is no different to normal Java bean configuration). However, instead of specifying the fully qualified classname of the class that is to be instantiated and configured by the container, you use the `<lang:language/>` element to define the dynamic language-backed bean.

Each of the supported languages has a corresponding `<lang:language/>` element:

- `<lang:jruby/>` (JRuby)
- `<lang:groovy/>` (Groovy)
- `<lang:bsh/>` (BeanShell)

The exact attributes and child elements that are available for configuration depends on exactly which language the bean has been defined in (the language-specific sections below provide the full lowdown on this).

24.3.1.2. Refreshable beans

One of the (if not *the*) most compelling value adds of the dynamic language support in Spring is the *'refreshable bean'* feature.

A refreshable bean is a dynamic-language-backed bean that with a small amount of configuration, a dynamic-language-backed bean can monitor changes in its underlying source file resource, and then reload itself when the dynamic language source file is changed (for example when a developer edits and saves changes to the file on the filesystem).

This allows a developer to deploy any number of dynamic language source files as part of an application, configure the Spring container to create beans backed by dynamic language source files (using the mechanisms described in this chapter), and then later, as requirements change or some other external factor comes into play, simply edit a dynamic language source file and have any change they make reflected in the bean that is backed by the changed dynamic language source file. There is no need to shut down a running application (or redeploy in the case of a web application). The dynamic-language-backed bean so amended will pick up the new state and logic from the changed dynamic language source file.

Note

Please note that this feature is *off* by default.

Let's take a look at an example to see just how easy it is to start using refreshable beans. To *turn on* the refreshable beans feature, you simply have to specify exactly *one* additional attribute on the `<lang:language/>` element of your bean definition. So if we stick with the example from earlier in this chapter, here's what we would change in the Spring XML configuration to effect refreshable beans:

```
<beans>

    <!-- this bean is now 'refreshable' due to the presence of the 'refresh-check-delay' attribute -->
    <lang:groovy id="messenger"
          refresh-check-delay="5000" <!-- switches refreshing on with 5 seconds between checks -->
```

```
          script-source="classpath:Messenger.groovy">
        <lang:property name="message" value="I Can Do The Frug" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

That really is all you have to do. The `'refresh-check-delay'` attribute defined on the `'messenger'` bean definition is the number of milliseconds after which the bean will be refreshed with any changes made to the underlying dynamic language source file. You can turn off the refresh behavior by assigning a negative value to the `'refresh-check-delay'` attribute. Remember that, by default, the refresh behavior is disabled. If you don't want the refresh behavior, then simply don't define the attribute.

If we then run the following application we can exercise the refreshable feature; please do excuse the *'jumping-through-hoops-to-pause-the-execution'* shenanigans in this next slice of code. The `System.in.read()` call is only there so that the execution of the program pauses while I (the author) go off and edit the underlying dynamic language source file so that the refresh will trigger on the dynamic-language-backed bean when the program resumes execution.

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.scripting.Messenger;

public final class Boot {

    public static void main(final String[] args) throws Exception {

        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Messenger messenger = (Messenger) ctx.getBean("messenger");
        System.out.println(messenger.getMessage());
        // pause execution while I go off and make changes to the source file...
        System.in.read();
        System.out.println(messenger.getMessage());
    }
}
```

Let's assume then, for the purposes of this example, that all calls to the `getMessage()` method of `Messenger` implementations have to be changed such that the message is surrounded by quotes. Below are the changes that I (the author) make to the `Messenger.groovy` source file when the execution of the program is paused.

```
package org.springframework.scripting

class GroovyMessenger implements Messenger {

    private String message = "Bingo"

    public String getMessage() {
        // change the implementation to surround the message in quotes
        return "'" + this.message + "'"
    }

    public void setMessage(String message) {
        this.message = message
    }
}
```

When the program executes, the output before the input pause will be `I Can Do The Frug`. After the change to the source file is made and saved, and the program resumes execution, the result of calling the `getMessage()` method on the dynamic-language-backed `Messenger` implementation will be `'I Can Do The Frug'` (notice the inclusion of the additional quotes).

It is important to understand that changes to a script will *not* trigger a refresh if the changes occur within the window of the `'refresh-check-delay'` value. It is equally important to understand that changes to the script are *not* actually 'picked up' until a method is called on the dynamic-language-backed bean. It is only when a method is called on a dynamic-language-backed bean that it checks to see if its underlying script source has changed. Any exceptions relating to refreshing the script (such as encountering a compilation error, or finding that the script file has been deleted) will result in a *fatal* exception being propagated to the calling code.

The refreshable bean behavior described above does *not* apply to dynamic language source files defined using the `<lang:inline-script/>` element notation (see the section entitled Section 24.3.1.3, “Inline dynamic language source files”). Additionally, it *only* applies to beans where changes to the underlying source file can actually be detected; for example, by code that checks the last modified date of a dynamic language source file that exists on the filesystem.

24.3.1.3. Inline dynamic language source files

The dynamic language support can also cater for dynamic language source files that are embedded directly in Spring bean definitions. More specifically, the `<lang:inline-script/>` element allows you to define dynamic language source immediately inside a Spring configuration file. An example will perhaps make the inline script feature crystal clear:

```
<lang:groovy id="messenger">
    <lang:inline-script>
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    String message
}
    </lang:inline-script>
    <lang:property name="message" value="I Can Do The Frug" />
</lang:groovy>
```

If we put to one side the issues surrounding whether it is good practice to define dynamic language source inside a Spring configuration file, the `<lang:inline-script/>` element can be useful in some scenarios. For instance, we might want to quickly add a Spring `Validator` implementation to a Spring MVC `Controller`. This is but a moment's work using inline source. (See the section entitled Section 24.4.2, “Scripted Validators” for such an example.)

Find below an example of defining the source for a JRuby-based bean directly in a Spring XML configuration file using the `inline:` notation. (Notice the use of the < characters to denote a `'<'` character. In such a case surrounding the inline source in a `<![CDATA[]]>` region might be better.)

```
<lang:jruby id="messenger" script-interfaces="org.springframework.scripting.Messenger">
    <lang:inline-script>
require 'java'

include_class 'org.springframework.scripting.Messenger'

class RubyMessenger &amp;lt; Messenger

 def setMessage(message)
  @@message = message
 end

 def getMessage
  @@message
 end

end
    </lang:inline-script>
    <lang:property name="message" value="Hello World!" />
```

```
</lang:jruby>
```

24.3.1.4. Understanding Constructor Injection in the context of dynamic-language-backed beans

There is one *very* important thing to be aware of with regard to Spring's dynamic language support. Namely, it is not (currently) possible to supply constructor arguments to dynamic-language-backed beans (and hence constructor-injection is not available for dynamic-language-backed beans). In the interests of making this special handling of constructors and properties 100% clear, the following mixture of code and configuration will *not* work.

```
// from the file 'Messenger.groovy'
package org.springframework.scripting.groovy;

import org.springframework.scripting.Messenger

class GroovyMessenger implements Messenger {

    GroovyMessenger() {}

    // this constructor is not available for Constructor Injection...
    GroovyMessenger(String message) {
        this.message = message;
    }

    String message

    String anotherMessage
}
```

```
<lang:groovy id="badMessenger"
    script-source="classpath:Messenger.groovy">

    <!-- this next constructor argument will *not* be injected into the GroovyMessenger -->
    <!--     in fact, this isn't even allowed according to the schema -->
    <constructor-arg value="This will *not* work" />

    <!-- only property values are injected into the dynamic-language-backed object -->
    <lang:property name="anotherMessage" value="Passed straight through to the dynamic-language-backed object" />

</lang>
```

In practice this limitation is not as significant as it first appears since setter injection is the injection style favored by the overwhelming majority of developers anyway (let's leave the discussion as to whether that is a good thing to another day).

24.3.2. JRuby beans

The JRuby library dependencies

The JRuby scripting support in Spring requires the following libraries to be on the classpath of your application. (The versions listed just happen to be the versions that the Spring team used in the development of the JRuby scripting support; you may well be able to use another version of a specific library.)

- jruby.jar
- cglib-nodep-2.1_3.jar

From the JRuby homepage...
“ JRuby is an 100% pure-Java implementation of the Ruby programming language. ”

In keeping with the Spring philosophy of offering choice, Spring's dynamic language support also supports beans defined in the JRuby language. The JRuby language is based on the quite intuitive Ruby language, and has support for inline regular expressions, blocks (closures), and a whole host of other features that do make solutions for some domain problems a whole lot easier to develop.

The implementation of the JRuby dynamic language support in Spring is interesting in that what happens is this: Spring creates a JDK dynamic proxy implementing all of the interfaces that are specified in the `'script-interfaces'` attribute value of the `<lang:ruby>` element (this is why you *must* supply at least one interface in the value of the attribute, and (accordingly) program to interfaces when using JRuby-backed beans).

Let us look at a fully working example of using a JRuby-based bean. Here is the JRuby implementation of the `Messenger` interface that was defined earlier in this chapter (for your convenience it is repeated below).

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

```
require 'java'

include_class 'org.springframework.scripting.Messenger'

class RubyMessenger < Messenger

 def setMessage(message)
  @@message = message
 end

 def getMessage
  @@message
 end

end

RubyMessenger.new # this last line is not essential (but see below)
```

And here is the Spring XML that defines an instance of the `RubyMessenger` JRuby bean.

```
<lang:jruby id="messageService"
            script-interfaces="org.springframework.scripting.Messenger"
            script-source="classpath:RubyMessenger.rb">

    <lang:property name="message" value="Hello World!" />

</lang:jruby>
```

Take note of the last line of that JRuby source (`'RubyMessenger.new'`). When using JRuby in the context of Spring's dynamic language support, you are encouraged to instantiate and return a new instance of the JRuby class that you want to use as a dynamic-language-backed bean as the result of the execution of your JRuby source. You can achieve this by simply instantiating a new instance of your JRuby class on the last line of the source file like so:

```
require 'java'

include_class 'org.springframework.scripting.Messenger'

# class definition same as above...
```

```
# instantiate and return a new instance of the RubyMessenger class
RubyMessenger.new
```

If you forget to do this, it is not the end of the world; this will however result in Spring having to trawl (reflectively) through the type representation of your JRuby class looking for a class to instantiate. In the grand scheme of things this will be so fast that you'll never notice it, but it is something that can be avoided by simply having a line such as the one above as the last line of your JRuby script. If you don't supply such a line, or if Spring cannot find a JRuby class in your script to instantiate then an opaque `ScriptCompilationException` will be thrown immediately after the source is executed by the JRuby interpreter. The key text that identifies this as the root cause of an exception can be found immediately below (so if your Spring container throws the following exception when creating your dynamic-language-backed bean and the following text is there in the corresponding stacktrace, this will hopefully allow you to identify and then easily rectify the issue):

```
org.springframework.scripting.ScriptCompilationException: Compilation of JRuby script returned ''
```

To rectify this, simply instantiate a new instance of whichever class you want to expose as a JRuby-dynamic-language-backed bean (as shown above). Please also note that you can actually define as many classes and objects as you want in your JRuby script; what is important is that the source file as a whole must return an object (for Spring to configure).

See the section entitled Section 24.4, “Scenarios” for some scenarios where you might want to use JRuby-based beans.

24.3.3. Groovy beans

The Groovy library dependencies

The Groovy scripting support in Spring requires the following libraries to be on the classpath of your application.

- `groovy-1.0.jar`
- `asm-2.2.2.jar`
- `antlr-2.7.6.jar`

From the Groovy homepage...
“ *Groovy is an agile dynamic language for the Java 2 Platform that has many of the features that people like so much in languages like Python, Ruby and Smalltalk, making them available to Java developers using a Java-like syntax.* ”

If you have read this chapter straight from the top, you will already have seen an example of a Groovy-dynamic-language-backed bean. Let's look at another example (again using an example from the Spring test suite).

Note

Groovy itself requires JDK 1.4+.

```
package org.springframework.scripting;
```

```
public interface Calculator {

    int add(int x, int y);
}
```

Here is an implementation of the `Calculator` interface in Groovy.

```
// from the file 'calculator.groovy'
package org.springframework.scripting.groovy

class GroovyCalculator implements Calculator {

    int add(int x, int y) {
        x + y
    }
}
```

```
<-- from the file 'beans.xml' -->
<beans>
    <lang:groovy id="calculator" script-source="classpath:calculator.groovy"/>
</beans>
```

Lastly, here is a small application to exercise the above configuration.

```
package org.springframework.scripting;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void Main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext("beans.xml");
        Calculator calc = (Calculator) ctx.getBean("calculator");
        System.out.println(calc.add(2, 8));
    }
}
```

The resulting output from running the above program will be (unsurprisingly) `10`. (Exciting example, huh? Remember that the intent is to illustrate the concept. Please consult the dynamic language showcase project for a more complex example, or indeed the section entitled Section 24.4, “Scenarios” later in this chapter).

It is important that you *do not* define more than one class per Groovy source file. While this is perfectly legal in Groovy, it is (arguably) a bad practice: in the interests of a consistent approach, you should (in the opinion of this author) respect the standard Java conventions of one (public) class per source file.

24.3.3.1. Customising Groovy objects via a callback

The `GroovyObjectCustomizer` interface is a callback that allows you to hook additional creation logic into the process of creating a Groovy-backed bean. For example, implementations of this interface could invoke any required initialization method(s), or set some default property values, or specify a custom `MetaClass`.

```
public interface GroovyObjectCustomizer {

   void customize(GroovyObject goo);
}
```

The Spring Framework will instantiate an instance of your Groovy-backed bean, and will then pass the created `GroovyObject` to the specified `GroovyObjectCustomizer` if one has been defined. You can do whatever you like with the supplied `GroovyObject` reference: it is expected that the setting of a custom `MetaClass` is what most folks will want to do with this callback, and you can see an example of doing that below.

```
public final class SimpleMethodTracingCustomizer implements GroovyObjectCustomizer {

   public void customize(GroovyObject goo) {
      DelegatingMetaClass metaClass = new DelegatingMetaClass(goo.getMetaClass()) {

         public Object invokeMethod(Object object, String methodName, Object[] arguments) {
            System.out.println("Invoking '" + methodName + "'.");
            return super.invokeMethod(object, methodName, arguments);
         }
      };
      metaClass.initialize();
      goo.setMetaClass(metaClass);
   }
}
```

A full discussion of meta-programming in Groovy is beyond the scope of the Spring reference manual. Consult the relevant section of the Groovy reference manual, or do a search online: there are plenty of articles concerning this topic. Actually making use of a `GroovyObjectCustomizer` is easy if you are using the Spring 2.0 namespace support.

```
<!-- define the GroovyObjectCustomizer just like any other bean -->
<bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer" />

   <!-- ... and plug it into the desired Groovy bean via the 'customizer-ref' attribute -->
   <lang:groovy id="calculator"
      script-source="classpath:org/springframework/scripting/groovy/Calculator.groovy"
      customizer-ref="tracingCustomizer" />
```

If you are not using the Spring 2.0 namespace support, you can still use the `GroovyObjectCustomizer` functionality.

```
<bean id="calculator" class="org.springframework.scripting.groovy.GroovyScriptFactory">
      <constructor-arg value="classpath:org/springframework/scripting/groovy/Calculator.groovy"/>
      <!-- define the GroovyObjectCustomizer (as an inner bean) -->
      <constructor-arg>
         <bean id="tracingCustomizer" class="example.SimpleMethodTracingCustomizer" />
      </constructor-arg>
</bean>

<bean class="org.springframework.scripting.support.ScriptFactoryPostProcessor"/>
```

24.3.4. BeanShell beans

The BeanShell library dependencies

The BeanShell scripting support in Spring requires the following libraries to be on the classpath of your application.

- `bsh-2.0b4.jar`
- `cglib-nodep-2.1_3.jar`

All of these libraries are available in the Spring-with-dependencies distribution of Spring (in addition to also being freely available on the web).

From the BeanShell homepage...
" *BeanShell is a small, free, embeddable Java source interpreter with dynamic language features, written in Java. BeanShell dynamically executes standard Java syntax and extends it with common scripting conveniences*

such as loose types, commands, and method closures like those in Perl and JavaScript. ”

In contrast to Groovy, BeanShell-backed bean definitions require some (small) additional configuration. The implementation of the BeanShell dynamic language support in Spring is interesting in that what happens is this: Spring creates a JDK dynamic proxy implementing all of the interfaces that are specified in the `'script-interfaces'` attribute value of the `<lang:bsh>` element (this is why you *must* supply at least one interface in the value of the attribute, and (accordingly) program to interfaces when using BeanShell-backed beans). This means that every method call on a BeanShell-backed object is going through the JDK dynamic proxy invocation mechanism.

Let's look at a fully working example of using a BeanShell-based bean that implements the `Messenger` interface that was defined earlier in this chapter (repeated below for your convenience).

```
package org.springframework.scripting;

public interface Messenger {

    String getMessage();
}
```

Here is the BeanShell 'implementation' (the term is used loosely here) of the `Messenger` interface.

```
String message;

String getMessage() {
    return message;
}

void setMessage(String aMessage) {
    message = aMessage;
}
```

And here is the Spring XML that defines an 'instance' of the above 'class' (again, the term is used very loosely here).

```
<lang:bsh id="messageService" script-source="classpath:BshMessenger.bsh"
    script-interfaces="org.springframework.scripting.Messenger">

    <lang:property name="message" value="Hello World!" />
</lang:bsh>
```

See the section entitled Section 24.4, “Scenarios” for some scenarios where you might want to use BeanShell-based beans.

24.4. Scenarios

The possible scenarios where defining Spring managed beans in a scripting language would be beneficial are, of course, many and varied. This section describes two possible use cases for the dynamic language support in Spring.

Please note that the Spring distribution ships with a showcase project for this dynamic language support in the relevant section of the Spring distribution. (A showcase project is a small project that is limited in scope to covering one particular aspect of the Spring Framework.)

24.4.1. Scripted Spring MVC Controllers

One group of classes that may benefit from using dynamic-language-backed beans is that of Spring MVC

controllers. In pure Spring MVC applications, the navigational flow through a web application is to a large extent determined by code encapsulated within your Spring MVC controllers. As the navigational flow and other presentation layer logic of a web application needs to be updated to respond to support issues or changing business requirements, it may well be easier to effect any such required changes by editing one or more dynamic language source files and seeing those changes being immediately reflected in the state of a running application.

Remember that in the lightweight architectural model espoused by projects such as Spring, you are typically aiming to have a really *thin* presentation layer, with all the meaty business logic of an application being contained in the domain and service layer classes. Developing Spring MVC controllers as dynamic-language-backed beans allows you to change presentation layer logic by simply editing and saving text files; any changes to such dynamic language source files will (depending on the configuration) automatically be reflected in the beans that are backed by dynamic language source files.

Note

In order to effect this automatic 'pickup' of any changes to dynamic-language-backed beans, you will have had to enable the 'refreshable beans' functionality. See the section entitle Section 24.3.1.2, “Refreshable beans” for a full treatment of this feature.

Find below an example of an `org.springframework.web.servlet.mvc.Controller` implemented using the Groovy dynamic language. This example is taken in part from the dynamic language support showcase project that is supplied with the Spring distribution; please do see the project in the `'samples/showcases/dynamvc/'` directory of the Spring distribution.

```
// from the file '/WEB-INF/groovy/FortuneController.groovy'
package org.springframework.showcase.fortune.web

import org.springframework.showcase.fortune.service.FortuneService
import org.springframework.showcase.fortune.domain.Fortune
import org.springframework.web.servlet.ModelAndView
import org.springframework.web.servlet.mvc.Controller

import javax.servlet.http.HttpServletRequest
import javax.servlet.http.HttpServletResponse

class FortuneController implements Controller {

    FortuneService fortuneService

    ModelAndView handleRequest(
            HttpServletRequest request, HttpServletResponse httpServletResponse) {

        return new ModelAndView("tell", "fortune", this.fortuneService.tellFortune())
    }
}
```

```
<lang:groovy id="fortune"
             refresh-check-delay="3000"
             script-source="/WEB-INF/groovy/FortuneController.groovy">
    <lang:property name="fortuneService" ref="fortuneService"/>
</lang:groovy>
```

24.4.2. Scripted Validators

Another area of application development with Spring that may benefit from the flexibility afforded by dynamic-language-backed beans is that of validation. It *may* be easier to express complex validation logic using a loosely typed dynamic language (that may also have support for inline regular expressions) as opposed to

regular Java.

Again, developing validators as dynamic-language-backed beans allows you to change validation logic by simply editing and saving a simple text file; any such changes will (depending on the configuration) automatically be reflected in the execution of a running application and would not require the restart of an application.

Note

Please note that in order to effect the automatic 'pickup' of any changes to dynamic-language-backed beans, you will have had to enable the 'refreshable beans' feature. See the section entitled Section 24.3.1.2, “Refreshable beans” for a full and detailed treatment of this feature.

Find below an example of a Spring `org.springframework.validation.Validator` implemented using the Groovy dynamic language. (See the section entitled Section 5.2, “Validation using Spring's `Validator` interface” for a discussion of the `Validator` interface.)

```
import org.springframework.validation.Validator
import org.springframework.validation.Errors
import org.springframework.beans.TestBean

class TestBeanValidator implements Validator {

    boolean supports(Class clazz) {
        return TestBean.class.isAssignableFrom(clazz)
    }

    void validate(Object bean, Errors errors) {
        if(bean.name?.trim()?.size() > 0) {
            return
        }
        errors.reject("whitespace", "Cannot be composed wholly of whitespace.")
    }
}
```

24.5. Bits and bobs

This last section contains some bits and bobs related to the dynamic language support.

24.5.1. AOP - advising scripted beans

It is possible to use the Spring AOP framework to advise scripted beans. The Spring AOP framework actually is unaware that a bean that is being advised might be a scripted bean, so all of the AOP use cases and functionality that you may be using or aim to use will work with scripted beans. There is just one (small) thing that you need to be aware of when advising scripted beans... you cannot use class-based proxies, you must use interface-based proxies.

You are of course not just limited to advising scripted beans... you can also write aspects themselves in a supported dynamic language and use such beans to advise other Spring beans. This really would be an advanced use of the dynamic language support though.

24.5.2. Scoping

In case it is not immediately obvious, scripted beans can of course be scoped just like any other bean. The

`scope` attribute on the various `<lang:language/>` elements allows you to control the scope of the underlying scripted bean, just as it does with a regular bean. (The default scope is singleton, just as it is with 'regular' beans.)

Find below an example of using the `scope` attribute to define a Groovy bean scoped as a prototype.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

    <lang:groovy id="messenger" script-source="classpath:Messenger.groovy" scope="prototype">
        <lang:property name="message" value="I Can Do The RoboCop" />
    </lang:groovy>

    <bean id="bookingService" class="x.y.DefaultBookingService">
        <property name="messenger" ref="messenger" />
    </bean>

</beans>
```

See the section entitled Section 3.4, “Bean scopes” in Chapter 3, *The IoC container* for a fuller discussion of the scoping support in the Spring Framework.

24.6. Further Resources

Find below links to further resources about the various dynamic languages described in this chapter.

- The JRuby homepage
- The Groovy homepage
- The BeanShell homepage

Some of the more active members of the Spring community have also added support for a number of additional dynamic languages above and beyond the ones covered in this chapter. While it is possible that such third party contributions may be added to the list of languages supported by the main Spring distribution, your best bet for seeing if your favourite scripting language is supported is the Spring Modules project.

Chapter 25. Annotations and Source Level Metadata Support

25.1. Introduction

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

We could add metadata to a method as follows:

```
/**
 * Normal comments here
 * @@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 of these examples use Jakarta Commons Attributes syntax.

Source-level metadata was introduced to the mainstream by XDoclet (in the Java world) and by the release of Microsoft's .NET platform, which uses source-level attributes to control transactions, pooling and other behavior.

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 used exclusively by EJB. While it is desirable to externalize some things from program source code, some important enterprise settings - notably transaction characteristics - arguably 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 parameters like transaction timeouts might change!).

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 view metadata primarily as a way of generating code such as EJB artefacts.

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

- **Standard Java Annotations**: the standard Java metadata implementation (developed as JSR-175 and available in Java 5). Spring has specific Java 5 annotations for transactional demarcation, JMX, and aspects (to be precise they are AspectJ annotations). However, since Spring supports both Java 1.3 and 1.4 a solution for said JVM versions is needed too. Spring metadata support provides such a solution.
- **XDoclet**: well-established solution, primarily intended for code generation.

- Various **open source attribute implementations**, for Java 1.3 and 1.4, of which Commons Attributes is the most complete implementation. All these require a special pre- or post-compilation step.

25.2. 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:

- Even though Java 5 provides metadata support at language level, there will still be value in providing such an abstraction:
  - Java 5 metadata is static. It is associated with a class at compile time, and cannot be changed in a deployed environment (annotation state can actually be changed at runtime using reflection, but doing so would really be a bad practice). There is a need for hierarchical metadata, providing the ability to override certain attribute values in deployment - for example, in an XML file.
  - Java 5 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.
  - 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*; forcing the use of Java 5 is not an option in such an important area.
- Current metadata APIs, such as Commons Attributes (used by Spring 1.0-1.2) are hard to test. Spring provides a simple metadata interface that is much easier to mock.

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 aimed to provide the subset of metadata required to provide effective declarative enterprise services in the vein of EJB or .NET, on Java 1.3+. As of Spring 1.2, analogous Java 5 annotations are also supported, as a direct alternative to Commons Attributes.

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

With most attribute providers, attribute classes are 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.

25.3. Annotations

The Spring Framework ships with a number of custom Java 5+ annotations.

25.3.1. @Required

The `@Required` annotation in the `org.springframework.beans.factory.annotation` package can be used to *mark* a property as being *'required-to-be-set'* (i.e. an annotated (setter) method of a class must be configured to be dependency injected with a value), else an `Exception` will be thrown by the container at runtime.

The best way to illustrate the usage of this annotation is to show an example:

```
public class SimpleMovieLister {

    // the SimpleMovieLister has a dependency on the MovieFinder
    private MovieFinder movieFinder;

    // a setter method so that the Spring container can 'inject' a MovieFinder
    @Required
    public void setMovieFinder(MovieFinder movieFinder) {
        this.movieFinder = movieFinder;
    }

    // business logic that actually 'uses' the injected MovieFinder is omitted...
}
```

Hopefully the above class definition reads easy on the eye. Any and all `BeanDefinitions` for the `SimpleMovieLister` class must be provided with a value.

Let's look at an example of some XML configuration that will **not** pass validation.

```
<bean id="movieLister" class="x.y.SimpleMovieLister">
    <!-- whoops, no MovieFinder is set (and this property is @Required) -->
</bean>
```

At runtime the following message will be generated by the Spring container (the rest of the stack trace has been truncated).

```
Exception in thread "main" java.lang.IllegalArgumentException:
    Property 'movieFinder' is required for bean 'movieLister'.
```

There is one last little (small, tiny) piece of Spring configuration that is required to actually *'switch on'* this behavior. Simply annotating the *'setter'* properties of your classes is not enough to get this behavior. You need to enable a component that is aware of the `@Required` annotation and that can process it appropriately.

This component is the `RequiredAnnotationBeanPostProcessor` class. This is a special `BeanPostProcessor` implementation that is `@Required`-aware and actually provides the *'blow up if this required property has not been set'* logic. It is *very* easy to configure; simply drop the following bean definition into your Spring XML configuration.

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor"/>
```

Finally, one can configure an instance of the `RequiredAnnotationBeanPostProcessor` class to look for *another* `Annotation` type. This is great if you already have your own `@Required`-style annotation. Simply plug it into the definition of a `RequiredAnnotationBeanPostProcessor` and you are good to go.

By way of an example, let's suppose you (or your organization / team) have defined an attribute called `@Mandatory`. You can make a `RequiredAnnotationBeanPostProcessor` instance `@Mandatory`-aware like so:

```
<bean class="org.springframework.beans.factory.annotation.RequiredAnnotationBeanPostProcessor">
    <property name="requiredAnnotationType" value="your.company.package.Mandatory"/>
</bean>
```

Here is the source code for the `@Mandatory` annotation. You will need to ensure that your custom annotation type is itself annotated with appropriate annotations for it's target and runtime retention policy.

```
package your.company.package;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Mandatory {
}
```

25.3.2. Other @Annotations in Spring

Annotations are also used in a number of other places throughout Spring. Rather than being described here, these annotations are described in that section or chapter of the reference documentation to which they are most relevant.

- Section 9.5.6, “Using `@Transactional`”
- Section 6.8.1, “Using AspectJ to dependency inject domain objects with Spring”
- Section 6.2, “@AspectJ support”

25.4. 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 2.1 (http://jakarta.apache.org/commons/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 a fully qualified name (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.* This is done 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 for storing the generated files:

```
<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 additional step, Commons Attributes will create an index of all the attributes defined on your sources, for efficient lookup at runtime. The step looks like this:

```
<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. Also, Commons Attributes is a static API, while Spring provides a metadata interface that you can easily mock.

25.5. 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.

25.5.1. Fundamentals

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

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

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
  <property name="transactionInterceptor" ref="txInterceptor" />
</bean>

<bean id="txInterceptor" class="org.springframework.transaction.interceptor.TransactionInterceptor">
  <property name="transactionManager" ref="transactionManager" />
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.interceptor.AttributesTransactionAttributeSource">
      <property name="attributes" ref="attributes" />
    </bean>
  </property>
</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 the auto-proxy creator and the advisor. Note that the actual bean names are not important; what matters is their class.

The bean definition of class `org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator` will automatically advise ("auto-proxy") 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, however, do know about attributes.

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

It is 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 it with another implementation of the `org.springframework.metadata.Attributes` interface to source attributes from a different source.

25.5.2. Declarative transaction management

The most common use of source-level attributes is to provide declarative transaction management. 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.

Please note that you can specify transaction attributes at either class or method level. Class-level attributes, if specified, will be "inherited" by all methods whereas method attributes will wholly override any class-level attributes.

25.5.3. Pooling

You can also enable pooling behavior via class-level attributes. Spring can apply this behavior 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)
 */
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">
  <property name="attributes" ref="attributes" />
</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 class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">
  <property name="customTargetSourceCreators">
    <list>
      <ref bean="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.

25.5.4. 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 behavior. 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.)

25.6. 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 of 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
 *
 * @@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 request, HttpServletResponse response) throws Exception {
        System.out.println("Bar Crunching c and d =" + cruncher.concatenate("c", "d"));
        return new ModelAndView("test");
    }
}
```

For this auto-mapping 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
 *
 * @@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 request, HttpServletResponse response) 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 a 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.)

In summary, 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.

Part V. Sample applications

This final part of the reference documentation covers the sample applications that come with the Spring Framework distribution.

Chapter 26. Showcase applications

26.1. Introduction

The Spring Framework distribution also ships with a number of so-called *showcase* applications. Each showcase application provides fully working examples, focused on demonstrating exactly one new Spring 2.0 feature at a time. The idea is that you can take the code in these showcases and experiment with it, as opposed to having to create your own small project to test out new Spring 2.0 features. The scope of these showcase applications is deliberately limited; the domain model (if there even is one) consists of maybe one or two classes, and typical enterprise concerns such as security, error-checking, and transactional integrity are omitted deliberately.

26.2. Spring MVC Controllers implemented in a dynamic language

This small application showcases implementing Spring MVC Controllers using the dynamic language support introduced in Spring 2.0.

The web application is *very* simplistic, because the intent is to convey the basics of the dynamic language support as applied to Spring MVC and pretty much nothing else.

There is one Groovy file in the application. It is called 'FortuneController.groovy' and it is located in the 'war/WEB-INF/groovy' folder. This Groovy script file is referenced by the 'fortune' bean in the 'war/WEB-INF/fortune-servlet.xml' Spring MVC configuration file.

You will notice that the 'fortune' bean is set as refreshable via the use of the 'refresh-check-delay' attribute on the <lang:groovy/> element. The value of this attribute is set to '3000' which means that changes to the 'FortuneController.groovy' file will be picked up after a delay of 3 seconds.

If you deploy the application to Tomcat (for example), you can then go into the exploded '/WEB-INF/groovy' folder and edit the 'FortuneController.groovy' file directly. Any such changes that you make will be picked up automatically and the 'fortune' bean will be reconfigured... all without having to stop, redeploy and restart the application. Try it yourself... now admittedly there is not a lot of complex logic in the 'FortuneController.groovy' file (which is good because Controllers in Spring MVC should be as thin as possible).

You could try returning a default Fortune instead of delegating to the injected FortuneService, or you could return a different logical view name, or (if you are feeling more ambitious) you could try creating a custom Groovy implementation of the FortuneService interface and try plugging that into the web application. Perhaps your custom Groovy FortuneService could access a web service to get some Fortunes, or apply some different randomizing logic to the returned Fortune, or whatever. The key point is that you will be able to make these changes without having to redeploy (or bounce) your application. This is a great boon with regard to rapid prototyping.

26.2.1. Build and deployment

The *samples/showcases/dynamvc* directory contains the web-app source. For deployment, it needs to be built with Apache Ant. The only requirements are JDK >=1.4 (it is Groovy that requires at a minimum JDK 1.4) and Ant >=1.5.

Run "build.bat" in this directory for available targets (e.g. "build.bat build", "build.bat warfile"). Note that to start Ant this way, you'll need an XML parser in your classpath (e.g. in "%JAVA_HOME%/jre/lib/ext"; included in JDK 1.4). You can use "warfile.bat" as a shortcut for WAR file creation. The WAR file will be created in the "dist" directory.

26.3. Implementing DAOs using `SimpleJdbcTemplate` and `@Repository`

This small project showcases using some of the Java5 features in Spring to implement DAOs with Hibernate and JDBC. This project is *very* simplistic, because the intent is to convey the basics of using the `SimpleJdbcTemplate` and the `@Repository` annotation and several other DAO-related features, but nothing else.

26.3.1. The domain

The domain in this sample application concerns itself with car parts. There are two domain classes: `Part` and `CarModel`. Using a `CarPartsInventory` car plants for example will be able to query for parts, update the stock of certain parts and add new parts.

26.3.2. The data access objects

Based on a `CarPartsInventory` interface there are 3 DAO implementations, each using a different style. Two are using Hibernate and the other is using JDBC. The `JdbcCarPartsInventoryImpl` uses JDBC and the `SimpleJdbcTemplate`. If you look closely at this DAO you will see that the Java5 features the `SimpleJdbcTemplate` uses significantly clean up your data access code.

The `TemplateHibernateCarPartsInventoryImpl` uses the `HibernateTemplate` to query for Parts and update part stock. This is nothing out of the ordinary if you're used to programming using Spring and Hibernate. The `PlainHibernateCarPartsInventoryImpl` however does not use the `HibernateTemplate` anymore. It uses the plain Hibernate3 API to query the session and the database for parts. Of course, the Hibernate3 API does not throw Spring `DataAccessExceptions` while this was originally one of the reasons to start using the `HibernateTemplate`. Spring 2.0 however adds an annotation that still allows you to get the same behavior. The `@Repository` annotation (if you look carefully at the `PlainHibernateCarPartsInventoryImpl`, you'll see it's marked as such) in combination with the `PersistenceExceptionTranslatorPostProcessor` automatically take care of translation Hibernate exception into Spring `DataAccessExceptions`.

26.3.3. Build

The *samples/showcases/java5-dao* directory contains the project's source. The project only contains unit tests that you can look at apart from the source code. To build and run the unit tests, you need to build with Apache Ant (or run the sample in your favorite IDE). Run `ant tests` using a Java5 VM (the project uses annotations and generics)

Appendix A. XML Schema-based configuration

A.1. Introduction

This appendix details the XML Schema-based configuration introduced in Spring 2.0.

> **DTD support?**
>
> Authoring Spring configuration files using the older DTD style is still fully supported.
>
> Nothing will break if you forego the use of the new XML Schema-based approach to authoring Spring XML configuration files. All that you lose out on is the opportunity to have more succinct and clearer configuration. Regardless of whether the XML configuration is DTD- or Schema-based, in the end it all boils down to the same object model in the container (namely one or more `BeanDefinition` instances).

The central motivation for moving to XML Schema based configuration files was to make Spring XML configuration easier. The *'classic'* `<bean/>`-based approach is good, but its generic-nature comes with a price in terms of configuration overhead.

From the Spring IoC containers point-of-view, *everything* is a bean. That's great news for the Spring IoC container, because if everything is a bean then everything can be treated in the exact same fashion. The same, however, is not true from a developer's point-of-view. The objects defined in a Spring XML configuration file are not all generic, vanilla beans. Usually, each bean requires some degree of specific configuration.

Spring 2.0's new XML Schema-based configuration addresses this issue. The `<bean/>` element is still present, and if you wanted to, you could continue to write the *exact same* style of Spring XML configuration using only `<bean/>` elements. The new XML Schema-based configuration does, however, make Spring XML configuration files substantially clearer to read. In addition, it allows you to express the intent of a bean definition.

The key thing to remember is that the new custom tags work best for infrastructure or integration beans: for example, AOP, collections, transactions, integration with 3rd-party frameworks such as Mule, etc., while the existing bean tags are best suited to application-specific beans, such as DAOs, service layer objects, validators, etc.

The examples included below will hopefully convince you that the inclusion of XML Schema support in Spring 2.0 was a good idea. The reception in the community has been encouraging; also, please note the fact that this new configuration mechanism is totally customisable and extensible. This means you can write your own domain-specific configuration tags that would better represent your application's domain; the process involved in doing so is covered in the appendix entitled Appendix B, *Extensible XML authoring*.

A.2. XML Schema-based configuration

A.2.1. Referencing the schemas

To switch over from the DTD-style to the new XML Schema-style, you need to make the following change.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
    "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>

<!-- <bean/> definitions here -->

</beans>
```

The equivalent file in the XML Schema-style would be...

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

Note

The `'xsi:schemaLocation'` fragment is not actually required, but can be included to reference a local copy of a schema (which can be useful during development).

The above Spring XML configuration fragment is boilerplate that you can copy and paste (!) and then plug `<bean/>` definitions into like you have always done. However, the entire point of switching over is to take advantage of the new Spring 2.0 XML tags since they make configuration easier. The section entitled Section A.2.2, “The `util` schema” demonstrates how you can start immediately by using some of the more common utility tags.

The rest of this chapter is devoted to showing examples of the new Spring XML Schema based configuration, with at least one example for every new tag. The format follows a before and after style, with a *before* snippet of XML showing the old (but still 100% legal and supported) style, followed immediately by an *after* example showing the equivalent in the new XML Schema-based style.

A.2.2. The `util` schema

First up is coverage of the `util` tags. As the name implies, the `util` tags deal with common, *utility* configuration issues, such as configuring collections, referencing constants, and suchlike.

To use the tags in the `util` schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the snippet below references the correct schema so that the tags in the `util` namespace are available to you.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-2.0.xsd">

<!-- <bean/> definitions here -->
```

```
</beans>
```

A.2.2.1. `<util:constant/>`

Before...

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
    class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `FieldRetrievingFactoryBean`, to set the value of the `'isolation'` property on a bean to the value of the `'java.sql.Connection.TRANSACTION_SERIALIZABLE'` constant. This is all well and good, but it is a tad verbose and (unneccessarily) exposes Spring's internal plumbing to the end user.

The following XML Schema-based version is more concise and clearly expresses the developer's intent (*'inject this constant value'*), and it just reads better.

```
<bean id="..." class="...">
  <property name="isolation">
    <util:constant static-field="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
  </property>
</bean>
```

A.2.2.1.1. Setting a bean property or constructor arg from a field value

`FieldRetrievingFactoryBean` is a `FactoryBean` which retrieves a `static` or non-static field value. It is typically used for retrieving `public` `static` `final` constants, which may then be used to set a property value or constructor arg for another bean.

Find below an example which shows how a `static` field is exposed, by using the `staticField` property:

```
<bean id="myField"
        class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
  <property name="staticField" value="java.sql.Connection.TRANSACTION_SERIALIZABLE"/>
</bean>
```

There is also a convenience usage form where the `static` field is specified as the bean name:

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
    class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

This does mean that there is no longer any choice in what the bean id is (so any other bean that refers to it will also have to use this longer name), but this form is very concise to define, and very convenient to use as an inner bean since the id doesn't have to be specified for the bean reference:

```
<bean id="..." class="...">
  <property name="isolation">
    <bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
          class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
  </property>
</bean>
```

It is also possible to access a non-static (instance) field of another bean, as described in the API documentation for the `FieldRetrievingFactoryBean` class.

Injecting enum values into beans as either property or constructor arguments is very easy to do in Spring, in that you don't actually have to *do* anything or know anything about the Spring internals (or even about classes such as the `FieldRetrievingFactoryBean`). Let's look at an example to see how easy injecting an enum value is; consider this JDK 5 enum:

```
package javax.persistence;

public enum PersistenceContextType {

    TRANSACTION,
    EXTENDED

}
```

Now consider a setter of type `PersistenceContextType`:

```
package example;

public class Client {

    private PersistenceContextType persistenceContextType;

    public void setPersistenceContextType(PersistenceContextType type) {
        this.persistenceContextType = type;
    }
}
```

.. and the corresponding bean definition:

```
<bean class="example.Client">
    <property name="persistenceContextType" value="TRANSACTION" />
</bean>
```

This works for classic type-safe emulated enums (on JDK 1.4 and JDK 1.3) as well; Spring will automatically attempt to match the string property value to a constant on the enum class.

A.2.2.2. `<util:property-path/>`

Before...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<bean id="testBean.age" class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

The above configuration uses a Spring `FactoryBean` implementation, the `PropertyPathFactoryBean`, to create a bean (of type `int`) called `'testBean.age'` that has a value equal to the `'age'` property of the `'testBean'` bean.

After...

```
<!-- target bean to be referenced by name -->
<bean id="testBean" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
```

```
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

<!-- will result in 10, which is the value of property 'age' of bean 'testBean' -->
<util:property-path id="name" path="testBean.age"/>
```

The value of the `'path'` attribute of the `<property-path/>` tag follows the form `'beanName.beanProperty'`.

A.2.2.2.1. Using `<util:property-path/>` to set a bean property or constructor-argument

`PropertyPathFactoryBean` is a `FactoryBean` that evaluates a property path on a given target object. The target object can be specified directly or via a bean name. This value may then be used in another bean definition as a property value or constructor argument.

Here's an example where a path is used against another bean, by name:

```
// target bean to be referenced by name
<bean id="person" class="org.springframework.beans.TestBean" scope="prototype">
  <property name="age" value="10"/>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="11"/>
    </bean>
  </property>
</bean>

// will result in 11, which is the value of property 'spouse.age' of bean 'person'
<bean id="theAge"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetBeanName" value="person"/>
  <property name="propertyPath" value="spouse.age"/>
</bean>
```

In this example, a path is evaluated against an inner bean:

```
<!-- will result in 12, which is the value of property 'age' of the inner bean -->
<bean id="theAge"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean">
  <property name="targetObject">
    <bean class="org.springframework.beans.TestBean">
      <property name="age" value="12"/>
    </bean>
  </property>
  <property name="propertyPath" value="age"/>
</bean>
```

There is also a shortcut form, where the bean name is the property path.

```
<!-- will result in 10, which is the value of property 'age' of bean 'person' -->
<bean id="person.age"
    class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
```

This form does mean that there is no choice in the name of the bean. Any reference to it will also have to use the same id, which is the path. Of course, if used as an inner bean, there is no need to refer to it at all:

```
<bean id="..." class="...">
  <property name="age">
    <bean id="person.age"
        class="org.springframework.beans.factory.config.PropertyPathFactoryBean"/>
  </property>
</bean>
```

The result type may be specifically set in the actual definition. This is not necessary for most use cases, but can be of use for some. Please see the Javadocs for more info on this feature.

A.2.2.3. `<util:properties/>`

Before...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<bean id="jdbcConfiguration" class="org.springframework.beans.factory.config.PropertiesFactoryBean">
  <property name="location" value="classpath:com/foo/jdbc-production.properties"/>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `PropertiesFactoryBean`, to instantiate a `java.util.Properties` instance with values loaded from the supplied `Resource` location).

After...

```
<!-- creates a java.util.Properties instance with values loaded from the supplied location -->
<util:properties id="jdbcConfiguration" location="classpath:com/foo/jdbc-production.properties"/>
```

A.2.2.4. `<util:list/>`

Before...

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<bean id="emails" class="org.springframework.beans.factory.config.ListFactoryBean">
  <property name="sourceList">
      <list>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </list>
  </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `ListFactoryBean`, to create a `java.util.List` instance initialized with values taken from the supplied `'sourceList'`.

After...

```
<!-- creates a java.util.List instance with values loaded from the supplied 'sourceList' -->
<util:list id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:list>
```

You can also explicitly control the exact type of `List` that will be instantiated and populated via the use of the `'list-class'` attribute on the `<util:list/>` element. For example, if we really need a `java.util.LinkedList` to be instantiated, we could use the following configuration:

```
<util:list id="emails" list-class="java.util.LinkedList">
    <value>jackshaftoe@vagabond.org</value>
    <value>eliza@thinkingmanscrumpet.org</value>
    <value>vanhoek@pirate.org</value>
    <value>d'Arcachon@nemesis.org</value>
</util:list>
```

If no `'list-class'` attribute is supplied, a `List` implementation will be chosen by the container.

Finally, you can also control the merging behavior using the `'merge'` attribute of the `<util:list/>` element; collection merging is described in more detail in the section entitled Section 3.3.3.4.1, “Collection merging”.

A.2.2.5. `<util:map/>`

Before...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<bean id="emails" class="org.springframework.beans.factory.config.MapFactoryBean">
  <property name="sourceMap">
      <map>
        <entry key="pechorin" value="pechorin@hero.org"/>
        <entry key="raskolnikov" value="raskolnikov@slums.org"/>
        <entry key="stavrogin" value="stavrogin@gov.org"/>
        <entry key="porfiry" value="porfiry@gov.org"/>
      </map>
  </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `MapFactoryBean`, to create a `java.util.Map` instance initialized with key-value pairs taken from the supplied `'sourceMap'`.

After...

```
<!-- creates a java.util.Map instance with values loaded from the supplied 'sourceMap' -->
<util:map id="emails">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

You can also explicitly control the exact type of `Map` that will be instantiated and populated via the use of the `'map-class'` attribute on the `<util:map/>` element. For example, if we really need a `java.util.TreeMap` to be instantiated, we could use the following configuration:

```
<util:map id="emails" map-class="java.util.TreeMap">
    <entry key="pechorin" value="pechorin@hero.org"/>
    <entry key="raskolnikov" value="raskolnikov@slums.org"/>
    <entry key="stavrogin" value="stavrogin@gov.org"/>
    <entry key="porfiry" value="porfiry@gov.org"/>
</util:map>
```

If no `'map-class'` attribute is supplied, a `Map` implementation will be chosen by the container.

Finally, you can also control the merging behavior using the `'merge'` attribute of the `<util:map/>` element; collection merging is described in more detail in the section entitled Section 3.3.3.4.1, “Collection merging”.

A.2.2.6. `<util:set/>`

Before...

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<bean id="emails" class="org.springframework.beans.factory.config.SetFactoryBean">
  <property name="sourceSet">
      <set>
        <value>pechorin@hero.org</value>
        <value>raskolnikov@slums.org</value>
        <value>stavrogin@gov.org</value>
        <value>porfiry@gov.org</value>
      </set>
```

```
  </property>
</bean>
```

The above configuration uses a Spring `FactoryBean` implementation, the `SetFactoryBean`, to create a `java.util.Set` instance initialized with values taken from the supplied `'sourceSet'`.

After...

```
<!-- creates a java.util.Set instance with values loaded from the supplied 'sourceSet' -->
<util:set id="emails">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

You can also explicitly control the exact type of `Set` that will be instantiated and populated via the use of the `'set-class'` attribute on the `<util:set/>` element. For example, if we really need a `java.util.TreeSet` to be instantiated, we could use the following configuration:

```
<util:set id="emails" set-class="java.util.TreeSet">
    <value>pechorin@hero.org</value>
    <value>raskolnikov@slums.org</value>
    <value>stavrogin@gov.org</value>
    <value>porfiry@gov.org</value>
</util:set>
```

If no `'set-class'` attribute is supplied, a `Set` implementation will be chosen by the container.

Finally, you can also control the merging behavior using the `'merge'` attribute of the `<util:set/>` element; collection merging is described in more detail in the section entitled Section 3.3.3.4.1, “Collection merging”.

A.2.3. The `jee` schema

The `jee` tags deal with JEE (Java Enterprise Edition)-related configuration issues, such as looking up a JNDI object and defining EJB references.

To use the tags in the `jee` schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the `jee` namespace are available to you.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/jee http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

A.2.3.1. `<jee:jndi-lookup/>` (simple)

Before...

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

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource"/>
```

A.2.3.2. `<jee:jndi-lookup/>` (with single JNDI environment setting)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <jee:environment>foo=bar</jee:environment>
</jee:jndi-lookup>
```

A.2.3.3. `<jee:jndi-lookup/>` (with multiple JNDI environment settings)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="jndiEnvironment">
        <props>
            <prop key="foo">bar</prop>
            <prop key="ping">pong</prop>
        </props>
    </property>
</bean>
```

After...

```
<jee:jndi-lookup id="simple" jndi-name="jdbc/MyDataSource">
    <!-- newline-separated, key-value pairs for the environment (standard Properties format) -->
    <jee:environment>
        foo=bar
        ping=pong
    </jee:environment>
</jee:jndi-lookup>
```

A.2.3.4. `<jee:jndi-lookup/>` (complex)

Before...

```
<bean id="simple" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/MyDataSource"/>
    <property name="cache" value="true"/>
    <property name="resourceRef" value="true"/>
    <property name="lookupOnStartup" value="false"/>
    <property name="expectedType" value="com.myapp.DefaultFoo"/>
    <property name="proxyInterface" value="com.myapp.Foo"/>
</bean>
```

After...

```
<jee:jndi-lookup id="simple"
             jndi-name="jdbc/MyDataSource"
             cache="true"
             resource-ref="true"
             lookup-on-startup="false"
             expected-type="com.myapp.DefaultFoo"
             proxy-interface="com.myapp.Foo"/>
```

A.2.3.5. `<jee:local-slsb/>` (simple)

The `<jee:local-slsb/>` tag configures a reference to an EJB Stateless SessionBean.

Before...

```
<bean id="simple"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/RentalServiceBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
</bean>
```

After...

```
<jee:local-slsb id="simpleSlsb" jndi-name="ejb/RentalServiceBean"
    business-interface="com.foo.service.RentalService"/>
```

A.2.3.6. `<jee:local-slsb/>` (complex)

```
<bean id="complexLocalEjb"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/RentalServiceBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
</bean>
```

After...

```
<jee:local-slsb id="complexLocalEjb"
    jndi-name="ejb/RentalServiceBean"
    business-interface="com.foo.service.RentalService"
    cache-home="true"
    lookup-home-on-startup="true"
    resource-ref="true">
```

A.2.3.7. `<jee:remote-slsb/>`

The `<jee:remote-slsb/>` tag configures a reference to a `remote` EJB Stateless SessionBean.

Before...

```
<bean id="complexRemoteEjb"
      class="org.springframework.ejb.access.SimpleRemoteStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="ejb/MyRemoteBean"/>
  <property name="businessInterface" value="com.foo.service.RentalService"/>
  <property name="cacheHome" value="true"/>
  <property name="lookupHomeOnStartup" value="true"/>
  <property name="resourceRef" value="true"/>
  <property name="homeInterface" value="com.foo.service.RentalService"/>
  <property name="refreshHomeOnConnectFailure" value="true"/>
```

```
</bean>
```

After...

```
<jee:remote-slsb id="complexRemoteEjb"
    jndi-name="ejb/MyRemoteBean"
    business-interface="com.foo.service.RentalService"
    cache-home="true"
    lookup-home-on-startup="true"
    resource-ref="true"
    home-interface="com.foo.service.RentalService"
    refresh-home-on-connect-failure="true">
```

A.2.4. The `lang` schema

The `lang` tags deal with exposing objects that have been written in a dynamic language such as JRuby or Groovy as beans in the Spring container.

These tags (and the dynamic language support) are comprehensively covered in the chapter entitled Chapter 24, *Dynamic language support*. Please do consult that chapter for full details on this support and the `lang` tags themselves.

In the interest of completeness, to use the tags in the `lang` schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the `lang` namespace are available to you.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:lang="http://www.springframework.org/schema/lang"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/lang http://www.springframework.org/schema/lang/spring-lang-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

A.2.5. The `tx` (transaction) schema

The `tx` tags deal with configuring all of those beans in Spring's comprehensive support for transactions. These tags are covered in the chapter entitled Chapter 9, *Transaction management*.

Tip

You are strongly encouraged to look at the `'spring-tx-2.0.xsd'` file that ships with the Spring distribution. This file is (of course), the XML Schema for Spring's transaction configuration, and covers all of the various tags in the `tx` namespace, including attribute defaults and suchlike. This file is documented inline, and thus the information is not repeated here in the interests of adhering to the DRY (Don't Repeat Yourself) principle.

In the interest of completeness, to use the tags in the `tx` schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the `tx` namespace are available to you.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:aop="http://www.springframework.org/schema/aop"
           xmlns:tx="http://www.springframework.org/schema/tx"
           xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-tx-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

Note

Often when using the tags in the `tx` namespace you will also be using the tags from the `aop` namespace (since the declarative transaction support in Spring is implemented using AOP). The above XML snippet contains the relevant lines needed to reference the `aop` schema so that the tags in the `aop` namespace are available to you.

A.2.6. The `aop` schema

The `aop` tags deal with configuring all things AOP in Spring: this includes Spring's own proxy-based AOP framework and Spring's integration with the AspectJ AOP framework. These tags are comprehensively covered in the chapter entitled Chapter 6, *Aspect Oriented Programming with Spring*.

In the interest of completeness, to use the tags in the `aop` schema, you need to have the following preamble at the top of your Spring XML configuration file; the emboldened text in the following snippet references the correct schema so that the tags in the `aop` namespace are available to you.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:aop="http://www.springframework.org/schema/aop"
           xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.0.xsd">

<!-- <bean/> definitions here -->

</beans>
```

A.2.7. The `tool` schema

The `tool` tags are for use when you want to add tooling-specific metadata to your custom configuration elements. This metadata can then be consumed by tools that are aware of this metadata, and the tools can then do pretty much whatever they want with it (validation, etc.).

The `tool` tags are not documented in this release of Spring as they are currently undergoing review. If you are a third party tool vendor and you would like to contribute to this review process, then do mail the Spring mailing list. The currently supported `tool` tags can be found in the file `'spring-tool-2.0.xsd'` in the `'src/org/springframework/beans/factory/xml'` directory of the Spring source distribution.

A.2.8. The `beans` schema

Last but not least we have the tags in the `beans` schema. These are the same tags that have been in Spring since

the very dawn of the framework. Examples of the various tags in the `beans` schema are not shown here because they are quite comprehensively covered in the section entitled Section 3.3.3, “Bean properties and constructor arguments detailed” (and indeed in that entire chapter).

One thing that is new to the beans tags themselves in Spring 2.0 is the idea of arbitrary bean metadata. In Spring 2.0 it is now possible to add zero or more key / value pairs to `<bean/>` XML definitions. What, if anything, is done with this extra metadata is totally up to your own custom logic (and so is typically only of use if you are writing your own custom tags as described in the appendix entitled Appendix B, *Extensible XML authoring*).

Find below an example of the `<meta/>` tag in the context of a surrounding `<bean/>` (please note that without any logic to interpret it the metadata is effectively useless as-is).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <bean id="foo" class="x.y.Foo">
        <meta key="cacheName" value="foo"/>
        <property name="name" value="Rick"/>
    </bean>

</beans>
```

In the case of the above example, you would assume that there is some logic that will consume the bean definition and set up some caching infrastructure using the supplied metadata.

A.3. Setting up your IDE

This final section documents the steps involved in setting up a number of popular Java IDEs to effect the easier editing of Spring's XML Schema-based configuration files. If your favourite Java IDE or editor is not included in the list of documented IDEs, then please do raise an issue and an example with your favorite IDE/editor *may* be included in the next release.

A.3.1. Setting up Eclipse

The following steps illustrate setting up Eclipse to be XSD-aware. The assumption in the following steps is that you already have an Eclipse project open (either a brand new project or an already existing one).

Note

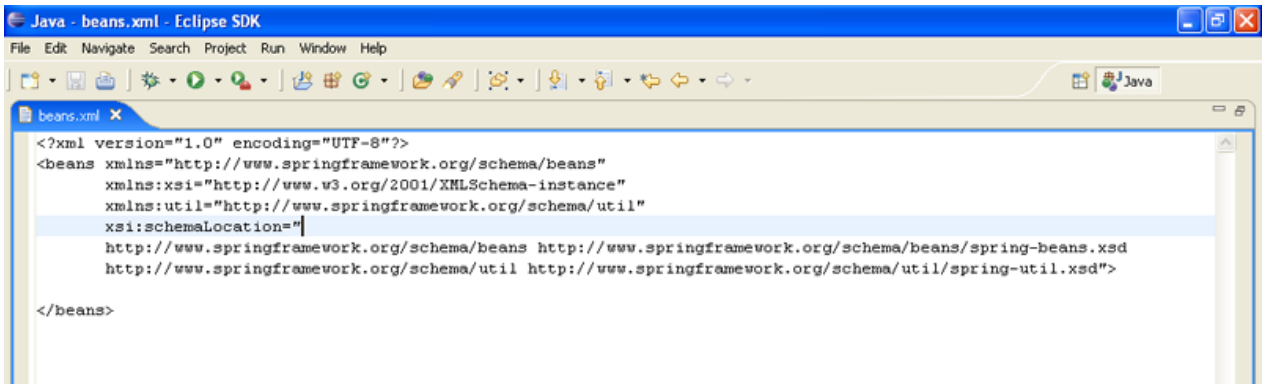
The following steps were created using Eclipse **3.2**. The setup will probably be the same (or similar) on an earlier or later version of Eclipse.

1. **Step One**

   Create a new XML file. You can name this file whatever you want. In the example below, the file is named `'context.xml'`. Copy and paste the following text into the file so that it matches the screenshot.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
```

```
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-2.0.xsd">

</beans>
```



2. **Step Two**

   As can be seen in the above screenshot (unless you have a customised version of Eclipse with the correct plugins) the XML file will be treated as plain text. There is no XML editing support out of the box in Eclipse, and as such there is not even any syntax highlighting of elements and attributes. To address this, you will have to install an XML editor plugin for Eclipse...

   Table A.1. Eclipse XML editors

XML Editor	Link
The Eclipse Web Tools Platform (WTP)	http://www.eclipse.org/webtools/
A list of Eclipse XML plugins	http://eclipse-plugins.2y.net/eclipse/plugins.jsp?category=XML

Contributing documentation...

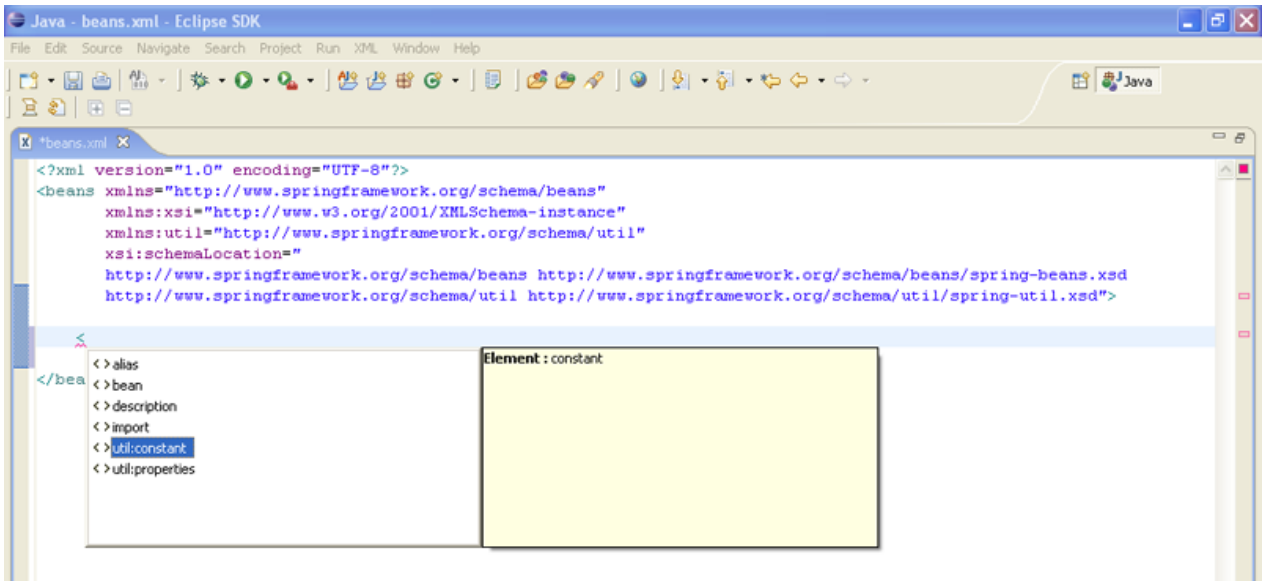
Patches showing how to configure an Eclipse XML editor are welcomed. Any such contributions are best submitted as patches via the Spring Framework JIRA Issue Tracker and *may* be featured in the next release.

   Unfortunately, precisely because there is no standard XML editor for Eclipse, there are (bar the one below) no further steps showing you how to configure XML Schema support in Eclipse... each XML editor plugin would require its very own dedicated section, and this is *Spring* reference documentation, not Eclipse XML editor documentation. You will have to read the documentation that comes with your XML editor plugin (good luck there) and figure it out for yourself.

3. **Step Three**

   However, if you are using the Web Tools Platform (WTP) for Eclipse, you don't need to do anything other than open a Spring XML configuration file using the WTP platform's XML editor. As can be seen in the screenshot below, you immediately get some slick IDE-level support for autocompleting tags and suchlike. The moral of this story is... download and install the WTP, because (quite simply, and to

paraphrase one of the Spring-WebFlow developers)... "*[WTP] rocks!*"



A.3.2. Setting up IntelliJ IDEA

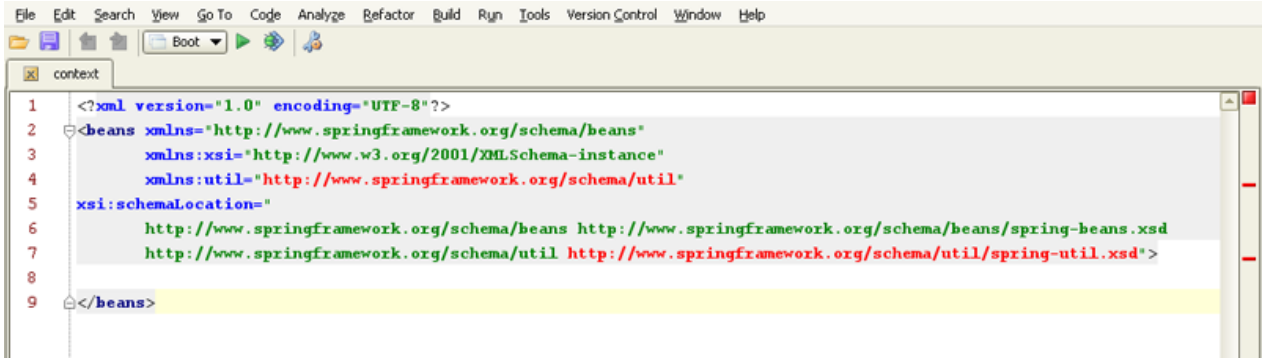
The following steps illustrate setting up the IntelliJ IDEA IDE to be XSD-aware. The assumption in the following steps is that you already have an IDEA project open (either a brand new project or an already existing one).

Repeat as required for setting up IDEA to reference the other Spring XSD files.

1. **Step One**

   Create a new XML file (you can name this file whatever you want). In the example below, the file is named `'context.xml'`. Copy and paste the following text into the file so that it matches the screenshot.

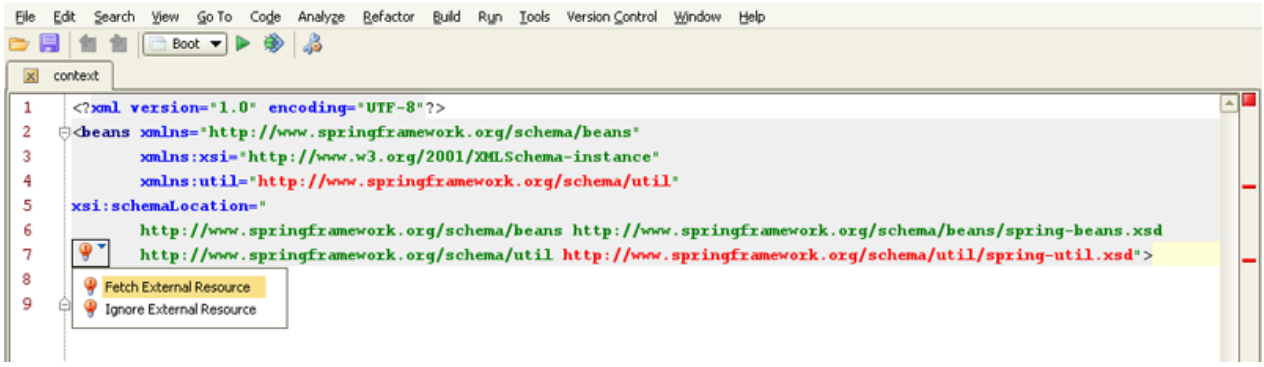
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="
       http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.
       http://www.springframework.org/schema/util http://www.springframework.org/schema/util/spring-util-2.0.xsd

</beans>
```



2. **Step Two**

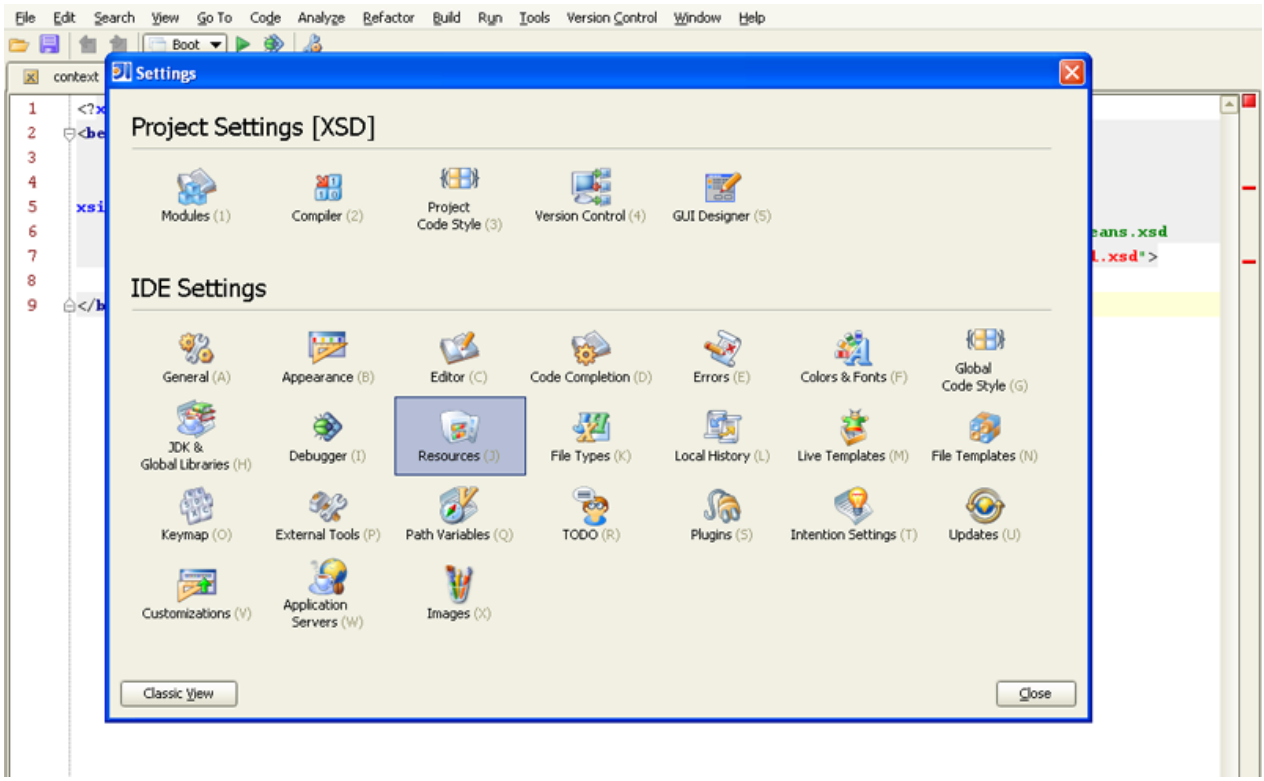
As can be seen in the above screenshot, the XML file has a number of nasty red contextual error markers. To rectify this, IDEA has to be made aware of the location of the referenced XSD namespace(s).

To do this, simply position the cursor over the squiggly red area (see the screenshot below); then press the **Alt-Enter** keystroke combination, and press the **Enter** key again when the popup becomes active to fetch the external resource.



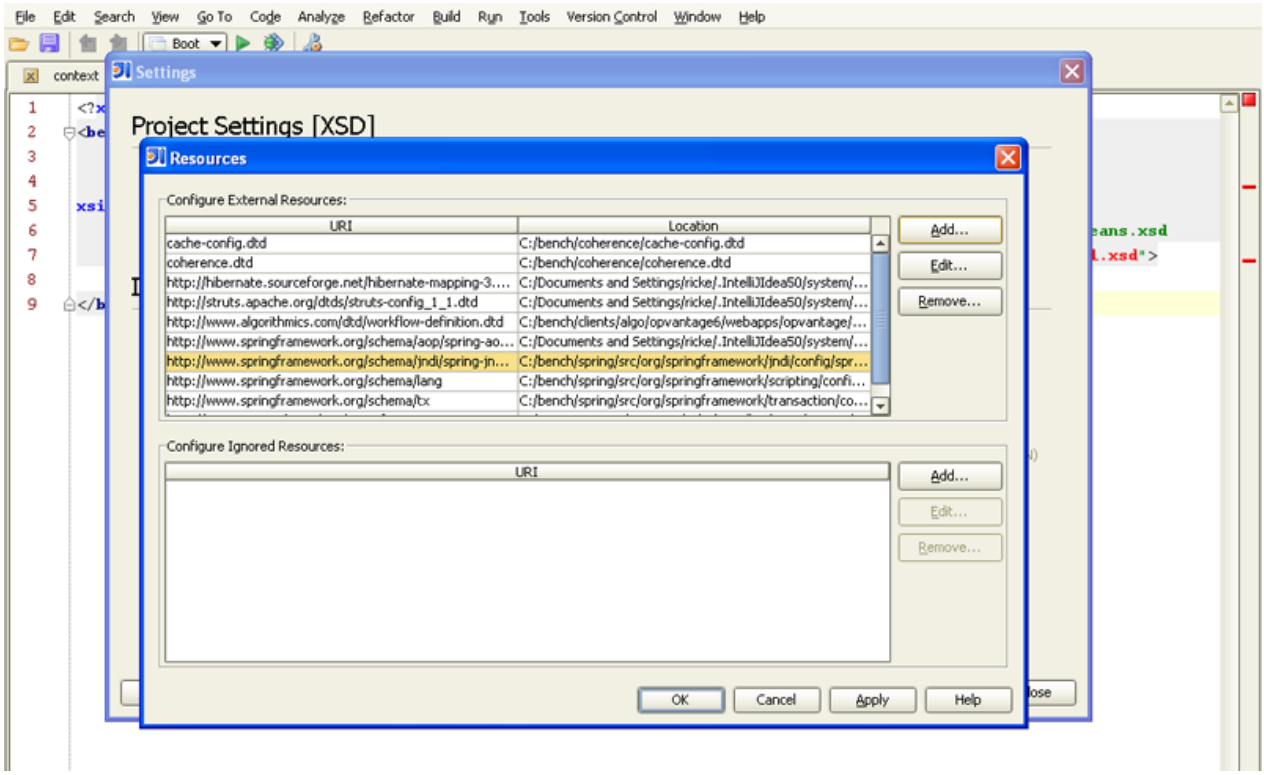
3. **Step Three**

If the external resource could not be fetched (maybe no active Internet connection is available), you can manually configure the resource to reference a local copy of the XSD file. Simply open up the `'Settings'` dialog (using the **Ctrl-A-S** keystroke combination or via the `'File|Settings'` menu), and click on the `'Resources'` button.



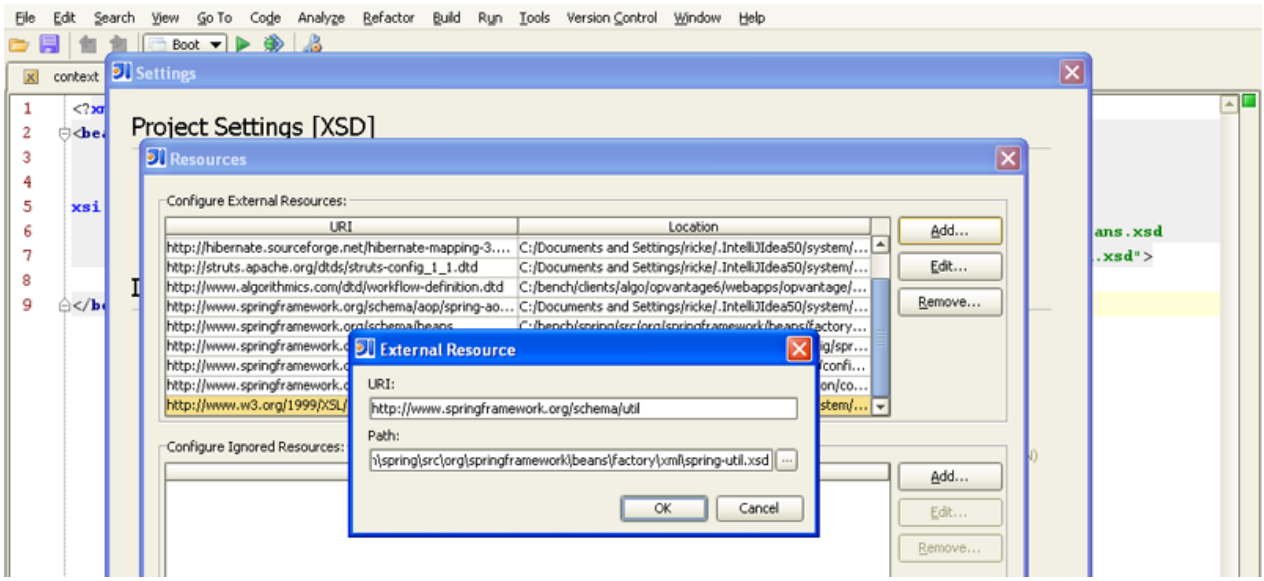
4. **Step Four**

As can be seen in the following screenshot, this will bring up a dialog that allows you to add an explicit reference to a local copy of the `util` schema file. (You can find all of the various Spring XSD files in the `'src'` directory of the Spring distribution.)



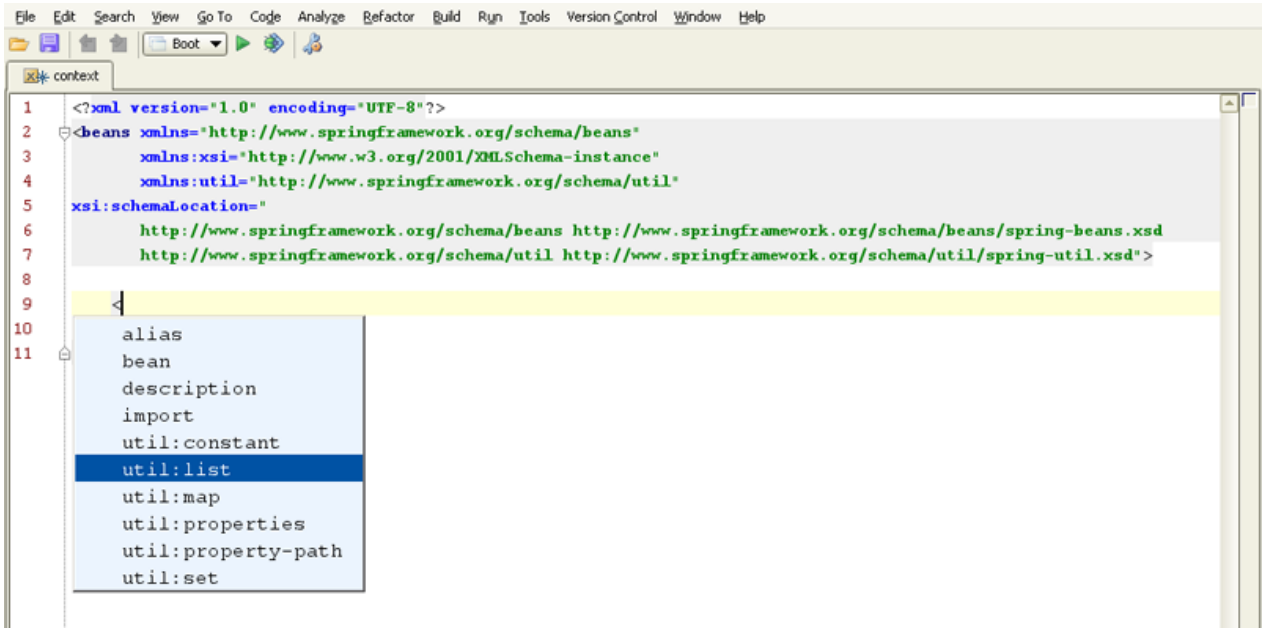
5. **Step Five**

Clicking the `'Add'` button will bring up another dialog that allows you to explicitly to associate a namespace URI with the path to the relevant XSD file. As can be seen in the following screenshot, the `'http://www.springframework.org/schema/util'` namespace is being associated with the file resource `'C:\bench\spring\src\org\springframework\beans\factory\xml\spring-util-2.0.xsd'`.



6. **Step Six**

Exiting out of the nested dialogs by clicking the `'OK'` button will then bring back the main editing window, and as can be seen in the following screenshot, the contextual error markers have disappeared; typing the `'<'` character into the editing window now also brings up a handy dropdown box that contains all of the imported tags from the `util` namespace.



A.3.3. Integration issues

This final section details integration issues that may arise when you switch over to using the above XSD-style for Spring 2.0 configuration.

This section is quite small at the moment (and hopefully it will stay that way). It has been included in the Spring documentation as a convenience to Spring users so that if you encounter an issue when switching over to the XSD-style in some specific environment you can refer to this section for the authoritative answer.

A.3.3.1. XML parsing errors in the Resin v.3 application server

If you are using the XSD-style for Spring 2.0 XML configuration and deploying to v.3 of Caucho's Resin application server, you will need to set some configuration options prior to startup so that an XSD-aware parser is available to Spring.

Please do read this resource, http://www.caucho.com/resin-3.0/xml/jaxp.xtp#xerces, for further details.

Appendix B. Extensible XML authoring

B.1. Introduction

Since version 2.0, Spring has featured a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. This section is devoted to detailing how you would go about writing your own custom XML bean definition parsers and integrating such parsers into the Spring IoC container.

To facilitate the authoring of configuration files using a schema-aware XML editor, Spring's extensible XML configuration mechanism is based on XML Schema. If you are not familiar with Spring's current XML configuration extensions that come with the standard Spring distribution, please first read the appendix entitled Appendix A, *XML Schema-based configuration*.

Creating new XML configuration extensions can be done by following these (relatively) simple steps:

1. Authoring an XML schema to describe your custom element(s).
2. Coding a custom `NamespaceHandler` implementation (this is an easy step, don't worry).
3. Coding one or more `BeanDefinitionParser` implementations (this is where the real work is done).
4. Registering the above artifacts with Spring (this too is an easy step).

What follows is a description of each of these steps. For the example, we will create an XML extension (a custom XML element) that allows us to configure objects of the type `SimpleDateFormat` (from the `java.text` package) in an easy manner. When we are done, we will be able to define bean definitions of type `SimpleDateFormat` like this:

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

(Don't worry about the fact that this example is very simple; much more detailed examples follow afterwards. The intent in this first simple example is to walk you through the basic steps involved.)

B.2. Authoring the schema

Creating an XML configuration extension for use with Spring's IoC container starts with authoring an XML Schema to describe the extension. What follows is the schema we'll use to configure `SimpleDateFormat` objects.

```
<!-- myns.xsd (inside package org/springframework/samples/xml) -->

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns="http://www.mycompany.com/schema/myns"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:beans="http://www.springframework.org/schema/beans"
    targetNamespace="http://www.mycompany.com/schema/myns"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">

   <xsd:import namespace="http://www.springframework.org/schema/beans"/>

   <xsd:element name="dateformat">
```

```
      <xsd:complexType>
         <xsd:complexContent>
            <xsd:extension base="beans:identifiedType">
               <xsd:attribute name="lenient" type="xsd:boolean"/>
               <xsd:attribute name="pattern" type="xsd:string" use="required"/>
            </xsd:extension>
         </xsd:complexContent>
      </xsd:complexType>
   </xsd:element>

</xsd:schema>
```

(The emphasized line contains an extension base for all tags that will be identifiable (meaning they have an `id` attribute that will be used as the bean identifier in the container). We are able to use this attribute because we imported the Spring-provided `'beans'` namespace.)

The above schema will be used to configure `SimpleDateFormat` objects, directly in an XML application context file using the `<myns:dateformat/>` element.

```
<myns:dateformat id="dateFormat"
    pattern="yyyy-MM-dd HH:mm"
    lenient="true"/>
```

Note that after we've created the infrastructure classes, the above snippet of XML will essentially be exactly the same as the following XML snippet. In other words, we're just creating a bean in the container, identified by the name `'dateFormat'` of type `SimpleDateFormat`, with a couple of properties set.

```
<bean id="dateFormat" class="java.text.SimpleDateFormat">
    <constructor-arg value="yyyy-HH-dd HH:mm"/>
    <property name="lenient" value="true"/>
</bean>
```

Note

The schema-based approach to creating configuration format allows for tight integration with an IDE that has a schema-aware XML editor. Using a properly authored schema, you can use autocompletion to have a user choose between several configuration options defined in the enumeration.

B.3. Coding a `NamespaceHandler`

In addition to the schema, we need a `NamespaceHandler` that will parse all elements of this specific namespace Spring encounters while parsing configuration files. The `NamespaceHandler` should in our case take care of the parsing of the `myns:dateformat` element.

The `NamespaceHandler` interface is pretty simple in that it features just three methods:

- `init()` - allows for initialization of the `NamespaceHandler` and will be called by Spring before the handler is used
- `BeanDefinition parse(Element, ParserContext)` - called when Spring encounters a top-level element (not nested inside a bean definition or a different namespace). This method can register bean definitions itself and/or return a bean definition.
- `BeanDefinitionHolder decorate(Node, BeanDefinitionHolder, ParserContext)` - called when Spring encounters an attribute or nested element of a different namespace. The decoration of one or more bean definitions is used for example with the out-of-the-box scopes Spring 2.0 supports. We'll start by highlighting a simple example, without using decoration, after which we will show decoration in a somewhat

more advanced example.

Although it is perfectly possible to code your own `NamespaceHandler` for the entire namespace (and hence provide code that parses each and every element in the namespace), it is often the case that each top-level XML element in a Spring XML configuration file results in a single bean definition (as in our case, where a single `<myns:dateformat/>` element results in a single `SimpleDateFormat` bean definition). Spring features a number of convenience classes that support this scenario. In this example, we'll make use the `NamespaceHandlerSupport` class:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class MyNamespaceHandler extends NamespaceHandlerSupport {

    public void init() {
        registerBeanDefinitionParser("dateformat", new SimpleDateFormatBeanDefinitionParser());
    }
}
```

The observant reader will notice that there isn't actually a whole lot of parsing logic in this class. Indeed... the `NamespaceHandlerSupport` class has a built in notion of delegation. It supports the registration of any number of `BeanDefinitionParser` instances, to which it will delegate to when it needs to parse an element in it's namespace. This clean separation of concerns allows a `NamespaceHandler` to handle the orchestration of the parsing of *all* of the custom elements in it's namespace, while delegating to `BeanDefinitionParsers` to do the grunt work of the XML parsing; this means that each `BeanDefinitionParser` will contain just the logic for parsing a single custom element, as we can see in the next step

B.4. Coding a `BeanDefinitionParser`

A `BeanDefinitionParser` will be used if the `NamespaceHandler` encounters an XML element of the type that has been mapped to the specific bean definition parser (which is `'dateformat'` in this case). In other words, the `BeanDefinitionParser` is responsible for parsing *one* distinct top-level XML element defined in the schema. In the parser, we'll have access to the XML element (and thus it's subelements too) so that we can parse our custom XML content, as can be seen in the following example:

```
package org.springframework.samples.xml;

import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.AbstractSingleBeanDefinitionParser;
import org.springframework.util.StringUtils;
import org.w3c.dom.Element;

import java.text.SimpleDateFormat;

public class SimpleDateFormatBeanDefinitionParser extends AbstractSingleBeanDefinitionParser { ❶

   protected Class getBeanClass(Element element) {
      return SimpleDateFormat.class; ❷
   }

   protected void doParse(Element element, BeanDefinitionBuilder bean) {
      // this will never be null since the schema explicitly requires that a value be supplied
      String pattern = element.getAttribute("pattern");
      bean.addConstructorArg(pattern);

      // this however is an optional property
      String lenient = element.getAttribute("lenient");
      if (StringUtils.hasText(lenient)) {
         bean.addPropertyValue("lenient", Boolean.valueOf(lenient));
      }
   }
}
```

❶ We use the Spring-provided `AbstractSingleBeanDefinitionParser` to handle a lot of the basic grunt work of creating a *single* `BeanDefinition`.

❷ We supply the `AbstractSingleBeanDefinitionParser` superclass with the type that our single `BeanDefinition` will represent.

In this simple case, this is all that we need to do. The creation of our single `BeanDefinition` is handled by the `AbstractSingleBeanDefinitionParser` superclass, as is the extraction and setting of the bean definition's unique identifier.

B.5. Registering the handler and the schema

The coding is finished! All that remains to be done is to somehow make the Spring XML parsing infrastructure aware of our custom element; we do this by registering our custom `namespaceHandler` and custom XSD file in two special purpose properties files. These properties files are both placed in a `'META-INF'` directory in your application, and can, for example, be distributed alongside your binary classes in a JAR file. The Spring XML parsing infrastructurewill automatically pick up your new extension by consuming these special properties files, the formats of which are detailed below.

B.5.1. `'META-INF/spring.handlers'`

The properties file called `'spring.handlers'` contains a mapping of XML Schema URIs to namespace handler classes. So for our example, we need to write the following:

```
http\://www.mycompany.com/schema/myns=org.springframework.samples.xml.MyNamespaceHandler
```

(The `':'` character is a valid delimiter in the Java properties format, and so the `':'` character in the URI needs to be escaped with a backslash.)

The first part (the key) of the key-value pair is the URI associated with your custom namespace extension, and needs to *match exactly* the value of the `'targetNamespace'` attribute as specified in your custom XSD schema.

B.5.2. `'META-INF/spring.schemas'`

The properties file called `'spring.schemas'` contains a mapping of XML Schema locations (referred to along with the schema declaration in XML files that use the schema as part of the `'xsi:schemaLocation'` attribute) to *classpath* resources. This file is needed to prevent Spring from absolutely having to use a default `EntityResolver` that requires Internet access to retrieve the schema file. If you specify the mapping in this properties file, Spring will search for the schema on the classpath (in this case `'myns.xsd'` in the `'org.springframework.samples.xml'` package):

```
http\://www.mycompany.com/schema/myns/myns.xsd=org/springframework/samples/xml/myns.xsd
```

The upshot of this is that you are encouraged to deploy your XSD file(s) right alongside the `NamespaceHandler` and `BeanDefinitionParser` classes on the classpath.

B.6. Using a custom extension in your Spring XML configuration

Using a custom extension that you yourself have implemented is no different from using one of the 'custom'

extensions that Spring provides straight out of the box. Find below an example of using the custom `<dateformat/>` element developed in the previous steps in a Spring XML configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:myns="http://www.mycompany.com/schema/myns"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.mycompany.com/schema/myns http://www.mycompany.com/schema/myns/myns.xsd">

   <!-- as a top-level bean -->
   <myns:dateformat id="defaultDateFormat" pattern="yyyy-MM-dd HH:mm" lenient="true"/>

   <bean id="jobDetailTemplate" abstract="true">
      <property name="dateFormat">
         <!-- as an inner bean -->
         <myns:dateformat pattern="HH:mm MM-dd-yyyy"/>
      </property>
   </bean>

</beans>
```

B.7. Meatier examples

Find below some much meatier examples of custom XML extensions.

B.7.1. Nesting custom tags within custom tags

This example illustrates how you might go about writing the various artifacts required to satisfy a target of the following configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:foo="http://www.foo.com/schema/component"
      xsi:schemaLocation="
http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://www.foo.com/schema/component http://www.foo.com/schema/component/component.xsd">

   <foo:component id="bionic-family" name="Bionic-1">
      <foo:component name="Sport-1"/>
      <foo:component name="Rock-1"/>
   </foo:component>

</beans>
```

The above configuration actually nests custom extensions within each other. The class that is actually configured by the above `<foo:component/>` element is the `Component` class (shown directly below). Notice how the `Component` class does *not* expose a setter method for the `'components'` property; this makes it hard (or rather impossible) to configure a bean definition for the `Component` class using setter injection.

```
package com.foo;

import java.util.ArrayList;
import java.util.List;

public class Component {

   private String name;
   private List components = new ArrayList();

   // mmm, there is no setter method for the 'components'
   public void addComponent(Component component) {
      this.components.add(component);
```

```
   }

   public List getComponents() {
      return components;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

The typical solution to this issue is to create a custom `FactoryBean` that exposes a setter property for the `'components'` property.

```
package com.foo;

import org.springframework.beans.factory.FactoryBean;

import java.util.Iterator;
import java.util.List;

public class ComponentFactoryBean implements FactoryBean {

   private Component parent;
   private List children;

   public void setParent(Component parent) {
      this.parent = parent;
   }

   public void setChildren(List children) {
      this.children = children;
   }

   public Object getObject() throws Exception {
      if (this.children != null && this.children.size() > 0) {
         for (Iterator it = children.iterator(); it.hasNext();) {
            Component childComponent = (Component) it.next();
            this.parent.addComponent(childComponent);
         }
      }
      return this.parent;
   }

   public Class getObjectType() {
      return Component.class;
   }

   public boolean isSingleton() {
      return true;
   }
}
```

This is all very well, and does work nicely, but exposes a lot of Spring plumbing to the end user. What we are going to do is write a custom extension that hides away all of this Spring plumbing. If we stick to the steps described previously, we'll start off by creating the XSD schema to define the structure of our custom tag.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/component"
         xmlns:xsd="http://www.w3.org/2001/XMLSchema"
         targetNamespace="http://www.foo.com/schema/component"
         elementFormDefault="qualified"
         attributeFormDefault="unqualified">

   <xsd:element name="component">
      <xsd:complexType>
         <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="component"/>
```

```
        </xsd:choice>
        <xsd:attribute name="id" type="xsd:ID"/>
        <xsd:attribute name="name" use="required" type="xsd:string"/>
      </xsd:complexType>
   </xsd:element>

</xsd:schema>
```

We'll then create a custom `NamespaceHandler`.

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class ComponentNamespaceHandler extends NamespaceHandlerSupport {

   public void init() {
      registerBeanDefinitionParser("component", new ComponentBeanDefinitionParser());
   }
}
```

Next up is the custom `BeanDefinitionParser`. Remember that what we are creating is a `BeanDefinition` describing a `ComponentFactoryBean`.

```
package com.foo;

import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.support.ManagedList;
import org.springframework.beans.factory.xml.AbstractBeanDefinitionParser;
import org.springframework.beans.factory.xml.ParserContext;
import org.springframework.util.xml.DomUtils;
import org.w3c.dom.Element;

import java.util.List;

public class ComponentBeanDefinitionParser extends AbstractBeanDefinitionParser {

   protected AbstractBeanDefinition parseInternal(Element element, ParserContext parserContext) {
      BeanDefinitionBuilder factory = BeanDefinitionBuilder.rootBeanDefinition(ComponentFactoryBean.class);
      BeanDefinitionBuilder parent = parseComponent(element);
      factory.addPropertyValue("parent", parent.getBeanDefinition());

      List childElements = DomUtils.getChildElementsByTagName(element, "component");
      if (childElements != null && childElements.size() > 0) {
         parseChildComponents(childElements, factory);
      }
      return factory.getBeanDefinition();
   }

   private static BeanDefinitionBuilder parseComponent(Element element) {
      BeanDefinitionBuilder component = BeanDefinitionBuilder.rootBeanDefinition(Component.class);
      component.addPropertyValue("name", element.getAttribute("name"));
      return component;
   }

   private static void parseChildComponents(List childElements, BeanDefinitionBuilder factory) {
      ManagedList children = new ManagedList(childElements.size());
      for (int i = 0; i < childElements.size(); ++i) {
         Element childElement = (Element) childElements.get(i);
         BeanDefinitionBuilder child = parseComponent(childElement);
         children.add(child.getBeanDefinition());
      }
      factory.addPropertyValue("children", children);
   }
}
```

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/component=com.foo.ComponentNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/component/component.xsd=com/foo/component.xsd
```

B.7.2. Custom attributes on 'normal' elements

Writing your own custom parser and the associated artifacts isn't hard, but sometimes it is not the right thing to do. Consider the scenario where you need to add metadata to already existing bean definitions. In this case you certainly don't want to have to go off and write your own entire custom extension; rather you just want to add an additional attribute to the existing bean definition element.

By way of another example, let's say that the service class that you are defining a bean definition for a service object that will (unknown to it) be accessing a clustered JCache, and you want to ensure that the named JCache instance is eagerly started within the surrounding cluster:

```
<bean id="checkingAccountService" class="com.foo.DefaultCheckingAccountService"
      jcache:cache-name="checking.account">
   <!-- other dependencies here... -->
</bean>
```

What we are going to do here is create another `BeanDefinition` when the `'jcache:cache-name'` attribute is parsed; this `BeanDefinition` will then initialize the named JCache for us. We will also modify the existing `BeanDefinition` for the `'checkingAccountService'` so that it will have a dependency on this new JCache-initializing `BeanDefinition`.

```
package com.foo;

public class JCacheInitializer {

   private String name;

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

   public void initialize() {
      // lots of JCache API calls to initialize the named cache...
   }
}
```

Now onto the custom extension. Firstly, the authoring of the XSD schema describing the custom attribute (quite easy in this case).

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>

<xsd:schema xmlns="http://www.foo.com/schema/jcache"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.foo.com/schema/jcache"
            elementFormDefault="qualified">

   <xsd:attribute name="cache-name" type="xsd:string"/>

</xsd:schema>
```

Next, the associated `NamespaceHandler`.

```
package com.foo;

import org.springframework.beans.factory.xml.NamespaceHandlerSupport;

public class JCacheNamespaceHandler extends NamespaceHandlerSupport {

   public void init() {
```

```
      super.registerBeanDefinitionDecoratorForAttribute("cache-name",
            new JCacheInitializingBeanDefinitionDecorator());
   }
}
```

Next, the parser. Note that in this case, because we are going to be parsing an XML attribute, we write a `BeanDefinitionDecorator` rather than a `BeanDefinitionParser`.

```
package com.foo;

import org.springframework.beans.factory.config.BeanDefinitionHolder;
import org.springframework.beans.factory.support.AbstractBeanDefinition;
import org.springframework.beans.factory.support.BeanDefinitionBuilder;
import org.springframework.beans.factory.xml.BeanDefinitionDecorator;
import org.springframework.beans.factory.xml.ParserContext;
import org.w3c.dom.Attr;
import org.w3c.dom.Node;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class JCacheInitializingBeanDefinitionDecorator implements BeanDefinitionDecorator {

   private static final String[] EMPTY_STRING_ARRAY = new String[0];

   public BeanDefinitionHolder decorate(
         Node source, BeanDefinitionHolder holder, ParserContext ctx) {
      String initializerBeanName = registerJCacheInitializer(source, ctx);
      createDependencyOnJCacheInitializer(holder, initializerBeanName);
      return holder;
   }

   private void createDependencyOnJCacheInitializer(BeanDefinitionHolder holder, String initializerBeanName) {
      AbstractBeanDefinition definition = ((AbstractBeanDefinition) holder.getBeanDefinition());
      String[] dependsOn = definition.getDependsOn();
      if (dependsOn == null) {
         dependsOn = new String[]{initializerBeanName};
      } else {
         List dependencies = new ArrayList(Arrays.asList(dependsOn));
         dependencies.add(initializerBeanName);
         dependsOn = (String[]) dependencies.toArray(EMPTY_STRING_ARRAY);
      }
      definition.setDependsOn(dependsOn);
   }

   private String registerJCacheInitializer(Node source, ParserContext ctx) {
      String cacheName = ((Attr) source).getValue();
      String beanName = cacheName + "-initializer";
      if (!ctx.getRegistry().containsBeanDefinition(beanName)) {
         BeanDefinitionBuilder initializer = BeanDefinitionBuilder.rootBeanDefinition(JCacheInitializer.class);
         initializer.addConstructorArg(cacheName);
         ctx.getRegistry().registerBeanDefinition(beanName, initializer.getBeanDefinition());
      }
      return beanName;
   }
}
```

Lastly, the various artifacts need to be registered with the Spring XML infrastructure.

```
# in 'META-INF/spring.handlers'
http\://www.foo.com/schema/jcache=com.foo.JCacheNamespaceHandler
```

```
# in 'META-INF/spring.schemas'
http\://www.foo.com/schema/jcache/jcache.xsd=com/foo/jcache.xsd
```

B.8. Further Resources

Find below links to further resources concerning XML Schema and the extensible XML support described in this chapter.

- The XML Schema Part 1: Structures Second Edition
- The XML Schema Part 2: Datatypes Second Edition

Appendix C. spring-beans-2.0.dtd

```
<!--
        Spring XML Beans DTD, version 2.0
        Authors: Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop

        This defines a simple and consistent way of creating a namespace
        of JavaBeans objects, managed by a Spring BeanFactory, read by
        XmlBeanDefinitionReader (with DefaultBeanDefinitionDocumentReader).

        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
        and/or constructor arguments.

        A bean instance can be a "singleton" (shared instance) or a "prototype"
        (independent instance). Further scopes can be provided by extended
        bean factories, for example in a web environment.

        References among beans are supported, that is, setting a JavaBean property
        or a constructor argument 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 of such inner bean definitions are effectively ignored:
        Inner beans are typically anonymous prototypes.

        There is also support for lists, sets, maps, and java.util.Properties
        as bean property types or constructor argument types.

        For simple purposes, this DTD is sufficient. As of Spring 2.0,
        XSD-based bean definitions are supported as more powerful alternative.

        XML documents that conform to this DTD should declare the following doctype:

        <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
                "http://www.springframework.org/dtd/spring-beans-2.0.dtd">
-->


<!--
        The document root. A document can contain bean definitions only,
        imports only, or a mixture of both (typically with imports first).
-->
<!ELEMENT beans (
        description?,
        (import | alias | 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-autowire (no | byName | byType | constructor | autodetect) "no">
<!ATTLIST beans default-dependency-check (none | objects | simple | all) "none">
<!ATTLIST beans default-init-method CDATA #IMPLIED>
<!ATTLIST beans default-destroy-method CDATA #IMPLIED>
<!ATTLIST beans default-merge (true | false) "false">

<!--
        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)>


<!--
        Specifies an XML bean definition resource to import.
-->
<!ELEMENT import EMPTY>
```

```
<!--
        The relative resource location of the XML bean definition file to import,
        for example "myImport.xml" or "includes/myImport.xml" or "../myImport.xml".
-->
<!ATTLIST import resource CDATA #REQUIRED>


<!--
        Defines an alias for a bean, which can reside in a different definition file.
-->
<!ELEMENT alias EMPTY>

<!--
        The name of the bean to define an alias for.
-->
<!ATTLIST alias name CDATA #REQUIRED>

<!--
        The alias name to define for the bean.
-->
<!ATTLIST alias alias CDATA #REQUIRED>

<!--
  Allows for arbitrary metadata to be attached to a bean definition.
-->
<!ELEMENT meta EMPTY>

<!--
  Specifies the key name of the metadata parameter being defined.
-->
<!ATTLIST meta key CDATA #REQUIRED>

<!--
  Specifies the value of the metadata parameter being defined as a String.
-->
<!ATTLIST meta value CDATA #REQUIRED>

<!--
        Defines a single (usually named) bean.

        A bean definition may contain nested tags for constructor arguments,
        property values, lookup methods, and replaced methods. Mixing constructor
        injection and setter injection on the same bean is explicitly supported.
-->
<!ELEMENT bean (
        description?,
        (meta | constructor-arg | property | lookup-method | replaced-method)*
)>

<!--
        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 is given, the bean class name is used as id
        (with an appended counter like "#2" if there is already a bean with that name).
-->
<!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, commas, or
        semi-colons (or indeed any mixture of the three).
-->
<!ATTLIST bean name CDATA #IMPLIED>

<!--
        Each bean definition must specify the fully qualified name of the class,
        except if it pure serves as parent for child bean definitions.
-->
<!ATTLIST bean class CDATA #IMPLIED>

<!--
        Optionally specify a parent bean definition.

        Will use the bean class of the parent if none specified, but can
```

```
        also override it. In the latter case, the child bean class must be
        compatible with the parent, i.e. accept the parent's property values
        and constructor argument values, if any.

        A child bean definition will inherit constructor argument values,
        property values and method overrides from the parent, with the option
        to add new values. If init method, destroy method, factory bean and/or factory
        method are specified, they will override the corresponding parent settings.

        The remaining settings will always be taken from the child definition:
        depends on, autowire mode, dependency check, scope, lazy init.
-->
<!ATTLIST bean parent CDATA #IMPLIED>

<!--
        The scope of this bean: typically "singleton" (one shared instance,
        which will be returned by all calls to getBean() with the id),
        or "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. Further scopes, such as "request" or "session",
        might be supported by extended bean factories (for example, in a
        web environment).

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.

        Inner bean definitions inherit the singleton status of their containing
        bean definition, unless explicitly specified: The inner bean will be a
        singleton if the containing bean is a singleton, and a prototype if
        the containing bean has any other scope.
-->
<!ATTLIST bean scope CDATA #IMPLIED>

<!--
        Is this bean "abstract", i.e. not meant to be instantiated itself but
        rather just serving as parent for concrete child bean definitions.
        Default is "false". Specify "true" to tell the bean factory to not try to
        instantiate that particular bean in any case.

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per abstract bean definition.
-->
<!ATTLIST bean abstract (true | false) #IMPLIED>

<!--
        If this bean should be lazily initialized.
        If false, it will get instantiated on startup by bean factories
        that perform eager initialization of singletons.

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.
-->
<!ATTLIST bean lazy-init (true | false | default) "default">

<!--
  Indicates whether or not this bean should be considered when looking
  for candidates to satisfy another beans autowiring requirements.
-->
<!ATTLIST bean autowire-candidate (true | false) #IMPLIED>

<!--
        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.
```

```
        If there is no matching bean by name, nothing special happens;
        use dependency-check="objects" to raise an error in that case.

        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 behavior can be combined with dependency
        checking, which will be performed after all autowiring has been completed.

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.
-->
<!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

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.
-->
<!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.

        Note: This attribute will not be inherited by child bean definitions.
        Hence, it needs to be specified per concrete bean definition.
-->
<!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 beans whose lifecycle is under full control
        of the factory - which is always the case for singletons, but not
        guaranteed for any other scope.
-->
<!ATTLIST bean destroy-method CDATA #IMPLIED>

<!--
        Optional attribute specifying the name of a factory method to use to
```

```
        create this object. Use constructor-arg elements to specify arguments
        to the factory method, if it takes arguments. Autowiring does not apply
        to factory methods.

        If the "class" attribute is present, the factory method will be a static
        method on the class specified by the "class" attribute on this bean
        definition. Often this will be the same class as that of the constructed
        object - for example, when the factory method is used as an alternative
        to a constructor. However, it may be on a different class. In that case,
        the created object will *not* be of the class specified in the "class"
        attribute. This is analogous to FactoryBean behavior.

        If the "factory-bean" attribute is present, the "class" attribute is not
        used, and the factory method will be an instance method on the object
        returned from a getBean call with the specified bean name. The factory
        bean may be defined as a singleton or a prototype.

        The factory method can have any number of arguments. Autowiring is not
        supported. Use indexed constructor-arg elements in conjunction with the
        factory-method attribute.

        Setter Injection can be used in conjunction with a factory method.
        Method Injection cannot, as the factory method returns an instance,
        which will be used when the container creates the bean.
-->
<!ATTLIST bean factory-method CDATA #IMPLIED>

<!--
        Alternative to class attribute for factory-method usage.
        If this is specified, no class attribute should be used.
        This should be set to the name of a bean in the current or
        ancestor factories that contains the relevant factory method.
        This allows the factory itself to be configured using Dependency
        Injection, and an instance (rather than static) method to be used.
-->
<!ATTLIST bean factory-bean CDATA #IMPLIED>

<!--
        Bean definitions can specify zero or more constructor arguments.
        This is an alternative to "autowire constructor".
        Arguments correspond to either a specific index of the constructor argument
        list or are supposed to be matched generically by type.

        Note: A single generic argument value will just be used once, rather than
        potentially matched multiple times (as of Spring 1.1).

        constructor-arg elements are also used in conjunction with the factory-method
        element to construct beans using static or instance factory methods.
-->
<!ELEMENT constructor-arg (
        description?,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        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>

<!--
  A short-cut alternative to a child element "ref bean=".
-->
<!ATTLIST constructor-arg ref CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "value".
-->
```

```
<!ATTLIST constructor-arg value 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?, meta*,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        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>

<!--
  A short-cut alternative to a child element "ref bean=".
-->
<!ATTLIST property ref CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "value".
-->
<!ATTLIST property value CDATA #IMPLIED>


<!--
        A lookup method causes the IoC container to override the given method and return
        the bean with the name given in the bean attribute. This is a form of Method Injection.
        It's particularly useful as an alternative to implementing the BeanFactoryAware
        interface, in order to be able to make getBean() calls for non-singleton instances
        at runtime. In this case, Method Injection is a less invasive alternative.
-->
<!ELEMENT lookup-method EMPTY>

<!--
        Name of a lookup method. This method should take no arguments.
-->
<!ATTLIST lookup-method name CDATA #IMPLIED>

<!--
        Name of the bean in the current or ancestor factories that the lookup method
        should resolve to. Often this bean will be a prototype, in which case the
        lookup method will return a distinct instance on every invocation. This
        is useful for single-threaded objects.
-->
<!ATTLIST lookup-method bean CDATA #IMPLIED>


<!--
        Similar to the lookup method mechanism, the replaced-method element is used to control
        IoC container method overriding: Method Injection. This mechanism allows the overriding
        of a method with arbitrary code.
-->
<!ELEMENT replaced-method (
        (arg-type)*
)>

<!--
        Name of the method whose implementation should be replaced by the IoC container.
        If this method is not overloaded, there's no need to use arg-type subelements.
        If this method is overloaded, arg-type subelements must be used for all
        override definitions for the method.
-->
<!ATTLIST replaced-method name CDATA #IMPLIED>

<!--
        Bean name of an implementation of the MethodReplacer interface in the current
        or ancestor factories. This may be a singleton or prototype bean. If it's
        a prototype, a new instance will be used for each method replacement.
        Singleton usage is the norm.
```

```
-->
<!ATTLIST replaced-method replacer CDATA #IMPLIED>

<!--
        Subelement of replaced-method identifying an argument for a replaced method
        in the event of method overloading.
-->
<!ELEMENT arg-type (#PCDATA)>

<!--
        Specification of the type of an overloaded method argument as a String.
        For convenience, this may be a substring of the FQN. E.g. all the
        following would match "java.lang.String":
        - java.lang.String
        - String
        - Str

        As the number of arguments will be checked also, this convenience can often
        be used to save typing.
-->
<!ATTLIST arg-type match CDATA #IMPLIED>


<!--
        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>
<!ATTLIST ref parent CDATA #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 supporting 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,
        potentially to be checked at runtime by bean factory 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>


<!--
        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 arbitrary target objects.

        Note that this is recommended for simple objects only.
        Configure more complex objects by populating JavaBean
        properties with references to other beans.
-->
<!ELEMENT value (#PCDATA)>

<!--
```

```
        The value tag can have an optional type attribute, to specify the
        exact type that the value should be converted to. Only needed
        if the type of the target property or constructor argument is
        too generic: for example, in case of a collection element.
-->
<!ATTLIST value type CDATA #IMPLIED>

<!--
        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)>


<!--
        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 the BeanFactory.
-->
<!ELEMENT list (
        (bean | ref | idref | value | null | list | set | map | props)*
)>

<!--
        Enable/disable merging for collections when using parent/child beans.
-->
<!ATTLIST list merge (true | false | default) "default">

<!--
        Specify the default Java type for nested values.
-->
<!ATTLIST list value-type CDATA #IMPLIED>


<!--
        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 | value | null | list | set | map | props)*
)>

<!--
        Enable/disable merging for collections when using parent/child beans.
-->
<!ATTLIST set merge (true | false | default) "default">

<!--
        Specify the default Java type for nested values.
-->
<!ATTLIST set value-type CDATA #IMPLIED>


<!--
        A Spring map is a mapping from a string key to object.
        Maps may be empty.
-->
<!ELEMENT map (
        (entry)*
)>

<!--
        Enable/disable merging for collections when using parent/child beans.
-->
<!ATTLIST map merge (true | false | default) "default">

<!--
        Specify the default Java type for nested entry keys.
-->
<!ATTLIST map key-type CDATA #IMPLIED>

<!--
        Specify the default Java type for nested entry values.
```

```
-->
<!ATTLIST map value-type CDATA #IMPLIED>

<!--
        A map entry can be an inner bean, ref, value, or collection.
        The key of the entry is given by the "key" attribute or child element.
-->
<!ELEMENT entry (
  key?,
        (bean | ref | idref | value | null | list | set | map | props)?
)>

<!--
        Each map element must specify its key as attribute or as child element.
        A key attribute is always a String value.
-->
<!ATTLIST entry key CDATA #IMPLIED>

<!--
  A short-cut alternative to a "key" element with a "ref bean=" child element.
-->
<!ATTLIST entry key-ref CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "value".
-->
<!ATTLIST entry value CDATA #IMPLIED>

<!--
  A short-cut alternative to a child element "ref bean=".
-->
<!ATTLIST entry value-ref CDATA #IMPLIED>

<!--
        A key element can contain an inner bean, ref, value, or collection.
-->
<!ELEMENT key (
        (bean | ref | idref | value | null | list | set | map | props)
)>


<!--
        Props elements differ from map elements in that values must be strings.
        Props may be empty.
-->
<!ELEMENT props (
        (prop)*
)>

<!--
        Enable/disable merging for collections when using parent/child beans.
-->
<!ATTLIST props merge (true | false | default) "default">

<!--
        Element content is the string value of the property.
        Note that whitespace is trimmed off to avoid unwanted whitespace
        caused by typical XML formatting.
-->
<!ELEMENT prop (#PCDATA)>

<!--
        Each property element must specify its key.
-->
<!ATTLIST prop key CDATA #REQUIRED>
```

Appendix D. spring.tld

D.1. Introduction

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). To help you implement views using Java Server Pages the Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages.

Please note that the various tags generated by this form tag library are compliant with the XHTML-1.0-Strict specification and attendant DTD.

This appendix describes the `spring.tld` tag library.

D.2. The `bind` tag

Provides BindStatus object for the given bind path. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.1. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
ignoreNestedPath	false	true
path	true	true

D.3. The `escapeBody` tag

Escapes its enclosed body content, applying HTML escaping and/or JavaScript escaping. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.2. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
javaScriptEscape	false	true

D.4. The `hasBindErrors` tag

Provides Errors instance in case of bind errors. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.3. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
name	true	true

D.5. The `htmlEscape` tag

Sets default HTML escape value for the current page. Overrides a "defaultHtmlEscape" context-param in web.xml, if any.

Table D.4. Attributes

Attribute	Required?	Runtime Expression?
defaultHtmlEscape	true	true

D.6. The `message` tag

Retrieves the message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.5. Attributes

Attribute	Required?	Runtime Expression?
arguments	false	true
argumentSeparator	false	true
code	false	true
htmlEscape	false	true
javaScriptEscape	false	true
message	false	true
scope	false	true
text	false	true
var	false	true

D.7. The `nestedPath` tag

Sets a nested path to be used by the bind tag's path.

Table D.6. Attributes

Attribute	Required?	Runtime Expression?
path	true	true

D.8. The `theme` tag

Retrieves the theme message with the given code, or text if code isn't resolvable. The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a "defaultHtmlEscape" context-param in web.xml).

Table D.7. Attributes

Attribute	Required?	Runtime Expression?
arguments	false	true

Attribute	Required?	Runtime Expression?
argumentSeparator	false	true
code	false	true
htmlEscape	false	true
javaScriptEscape	false	true
message	false	true
scope	false	true
text	false	true
var	false	true

D.9. The `transform` tag

Provides transformation of variables to Strings, using an appropriate custom PropertyEditor from BindTag (can only be used inside BindTag). The HTML escaping flag participates in a page-wide or application-wide setting (i.e. by HtmlEscapeTag or a 'defaultHtmlEscape' context-param in web.xml).

Table D.8. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
scope	false	true
value	true	true
var	false	true

Appendix E. spring-form.tld

E.1. Introduction

One of the view technologies you can use with the Spring Framework is Java Server Pages (JSPs). To help you implement views using Java Server Pages the Spring Framework provides you with some tags for evaluating errors, setting themes and outputting internationalized messages.

Please note that the various tags generated by this form tag library are compliant with the XHTML-1.0-Strict specification and attendant DTD.

This appendix describes the `spring-form.tld` tag library.

E.2. The `checkbox` tag

Renders an HTML 'input' tag with type 'checkbox'.

Table E.1. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
cssClass	false	true
cssErrorClass	false	true

Attribute	Required?	Runtime Expression?
cssStyle	false	true
dir	false	true
disabled	false	true
htmlEscape	false	true
id	false	true
lang	false	true
onblur	false	true
onchange	false	true
onclick	false	true
ondblclick	false	true
onfocus	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
path	true	true
tabindex	false	true

Attribute	Required?	Runtime Expression?
title	false	true
value	false	true

E.3. The `errors` tag

Renders field errors in an HTML 'span' tag.

Table E.2. Attributes

Attribute	Required?	Runtime Expression?
cssClass	false	true
cssStyle	false	true
delimiter	false	true
dir	false	true
element	false	true
htmlEscape	false	true
id	false	true
lang	false	true
onclick	false	true
ondblclick	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true

Attribute	Required?	Runtime Expression?
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
path	false	true
tabindex	false	true
title	false	true

E.4. The `form` tag

Renders an HTML 'form' tag and exposes a binding path to inner tags for binding.

Table E.3. Attributes

Attribute	Required?	Runtime Expression?
action	false	true
commandName	false	true
cssClass	false	true
cssStyle	false	true
dir	false	true
enctype	false	true
htmlEscape	false	true
id	false	true
lang	false	true
method	false	true
name	false	true

Attribute	Required?	Runtime Expression?
onclick	false	true
ondblclick	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
onreset	false	true
onsubmit	false	true
title	false	true

E.5. The `hidden` tag

Renders an HTML 'input' tag with type 'hidden' using the bound value.

Table E.4. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
id	false	true
path	true	true

E.6. The `input` tag

Renders an HTML 'input' tag with type 'text' using the bound value.

Table E.5. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
alt	false	true
autocomplete	false	true
cssClass	false	true
cssErrorClass	false	true
cssStyle	false	true
dir	false	true
disabled	false	true
htmlEscape	false	true
id	false	true
lang	false	true
maxlength	false	true
onblur	false	true
onchange	false	true
onclick	false	true
ondblclick	false	true
onfocus	false	true
onkeydown	false	true
onkeypress	false	true

Attribute	Required?	Runtime Expression?
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
onselect	false	true
path	true	true
readonly	false	true
size	false	true
tabindex	false	true
title	false	true

E.7. The `label` tag

Renders a form field label in an HTML 'label' tag.

Table E.6. Attributes

Attribute	Required?	Runtime Expression?
cssClass	false	true
cssErrorClass	false	true
cssStyle	false	true
dir	false	true
for	false	true

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
id	false	true
lang	false	true
onclick	false	true
ondblclick	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
path	true	true
tabindex	false	true
title	false	true

E.8. The `option` tag

Renders a single HTML 'option'. Sets 'selected' as appropriate based on bound value.

Table E.7. Attributes

Attribute	Required?	Runtime Expression?
disabled	false	true

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
label	false	true
value	true	true

E.9. The `options` tag

Renders a list of HTML 'option' tags. Sets 'selected' as appropriate based on bound value.

Table E.8. Attributes

Attribute	Required?	Runtime Expression?
htmlEscape	false	true
itemLabel	false	true
items	true	true
itemValue	false	true

E.10. The `password` tag

Renders an HTML 'input' tag with type 'password' using the bound value.

Table E.9. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
alt	false	true
autocomplete	false	true
cssClass	false	true
cssErrorClass	false	true
cssStyle	false	true

Attribute	Required?	Runtime Expression?
dir	false	true
disabled	false	true
htmlEscape	false	true
id	false	true
lang	false	true
maxlength	false	true
onblur	false	true
onchange	false	true
onclick	false	true
ondblclick	false	true
onfocus	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
onselect	false	true
path	true	true

Attribute	Required?	Runtime Expression?
readonly	false	true
showPassword	false	true
size	false	true
tabindex	false	true
title	false	true

E.11. The `radiobutton` tag

Renders an HTML 'input' tag with type 'radio'.

Table E.10. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
cssClass	false	true
cssErrorClass	false	true
cssStyle	false	true
dir	false	true
disabled	false	true
htmlEscape	false	true
id	false	true
lang	false	true
onblur	false	true
onchange	false	true
onclick	false	true

Attribute	Required?	Runtime Expression?
ondblclick	false	true
onfocus	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
path	true	true
tabindex	false	true
title	false	true
value	false	true

E.12. The `select` tag

Renders an HTML 'select' element. Supports databinding to the selected option.

Table E.11. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
cssClass	false	true
cssErrorClass	false	true

Attribute	Required?	Runtime Expression?
cssStyle	false	true
dir	false	true
disabled	false	true
htmlEscape	false	true
id	false	true
itemLabel	false	true
items	false	true
itemValue	false	true
lang	false	true
multiple	false	true
onblur	false	true
onchange	false	true
onclick	false	true
ondblclick	false	true
onfocus	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true

Attribute	Required?	Runtime Expression?
onmouseup	false	true
path	true	true
size	false	true
tabindex	false	true
title	false	true

E.13. The `textarea` tag

Renders an HTML 'textarea'.

Table E.12. Attributes

Attribute	Required?	Runtime Expression?
accesskey	false	true
cols	false	true
cssClass	false	true
cssErrorClass	false	true
cssStyle	false	true
dir	false	true
disabled	false	true
htmlEscape	false	true
id	false	true
lang	false	true
onblur	false	true
onchange	false	true

Attribute	Required?	Runtime Expression?
onclick	false	true
ondblclick	false	true
onfocus	false	true
onkeydown	false	true
onkeypress	false	true
onkeyup	false	true
onmousedown	false	true
onmousemove	false	true
onmouseout	false	true
onmouseover	false	true
onmouseup	false	true
onselect	false	true
path	true	true
readonly	false	true
rows	false	true
tabindex	false	true
title	false	true