java/j2ee Application Framework

Reference Documentation

Version 1.2.7

(Work in progress)

Copyright (c) 2004-2006 Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Rob Harrop, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet

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

Table of Contents

Preface
1. Introduction
1.1. Overview ... 1
1.2. Usage scenarios ... 2
2. Background information
2.1. Inversion of Control / Dependency Injection ... 5
3. Beans, BeanFactory and the ApplicationContext
3.1. Introduction ... 6
3.2. BeanFactory and BeanDefinitions - the basics ... 6
3.2.1. The BeanFactory ... 6
3.2.2. The BeanDefinition ... 8
3.2.3. The bean class ... 8
3.2.4. The bean identifiers (id and name) ... 10
3.2.5. To singleton or not to singleton ... 10
3.3. Properties, collaborators, autowiring and dependency checking ... 11
3.3.1. Setting bean properties and collaborators ... 11
3.3.2. Constructor Argument Resolution ... 14
3.3.3. Bean properties and constructor arguments detailed ... 15
3.3.4. Method Injection ... 19
3.3.5. Using depends-on ... 21
3.3.6. Autowiring collaborators ... 22
3.3.7. Checking for dependencies ... 23
3.4. Customizing the nature of a bean ... 23
3.4.1. Lifecycle interfaces ... 24
3.4.2. Knowing who you are ... 25
3.4.3. FactoryBean ... 26
3.5. Abstract and child bean definitions ... 26
3.6. Interacting with the BeanFactory ... 27
3.6.1. Obtaining a FactoryBean, not its product ... 28
3.7. Customizing beans with BeanPostProcessors ... 28
3.8. Customizing bean factories with BeanFactoryPostProcessors ... 28
3.8.1. The PropertyPlaceholderConfigurer ... 29
3.8.2. The PropertyOverrideConfigurer ... 30
3.9. Registering additional custom PropertyEditors ... 30
3.10. Using the alias element to add aliases for existing beans ... 31
3.11. Introduction to the ApplicationContext ... 32
3.12. Added functionality of the ApplicationContext ... 32
3.12.1. Using the MessageSource ... 32
3.12.2. Propagating events ... 33
3.12.3. Low-level resources and the application context ... 34
3.13. Customized behavior in the ApplicationContext ... 35
3.13.1. ApplicationContextAware marker interface ... 35
3.13.2. The BeanPostProcessor ... 35
3.13.3. The BeanFactoryPostProcessor ... 35

3.13.4. The PropertyPlaceholderConfigurer 36
3.14. Registering additional custom PropertyEditors 36
3.15. Setting a bean property or constructor arg from a property expression 37
3.16. Setting a bean property or constructor arg from a field value 38
3.17. Invoking another method and optionally using the return value. 38
3.18. Importing Bean Definitions from One File Into Another 39
3.19. Creating an ApplicationContext from a web application 40
3.20. Glue code and the evil singleton 41
3.20.1. Using SingletonBeanFactoryLocator and ContextSingletonBeanFactoryLocator 41
4. Abstracting Access to Low-Level Resources ....
4.1. Overview 43
4.2. The Resource interface 43
4.3. Built-in Resource implementations 44
4.3.1. UrlResource 44
4.3.2. ClassPathResource 44
4.3.3. FileSystemResource 45
4.3.4. ServletContextResource 45
4.3.5. InputStreamResource 45
4.3.6. ByteArrayResource 45
4.4. The ResourceLoader Interface 45
4.5. The ResourceLoaderAware interface 46
4.6. Setting Resources as properties 46
4.7. Application contexts and Resource paths 47
4.7.1. Constructing application contexts 47
4.7.2. The classpath*: prefix 47
4.7.3. Unexpected application context handling of FileSystemResource absolute paths 48
5. PropertyEditors, data binding, validation and the BeanWrapper ....
5.1. Introduction 50
5.2. Binding data using the DataBinder 50
5.3. Bean manipulation and the BeanWrapper 50
5.3.1. Setting and getting basic and nested properties 51
5.3.2. Built-in PropertyEditors, converting types 52
5.3.3. Other features worth mentioning 54
5.4. Validation using Spring's Validator interface 54
5.5. The Errors interface 55
5.6. Resolving codes to error messages 55
6. Spring AOP: Aspect Oriented Programming with Spring ....
6.1. Concepts 56
6.1.1. AOP concepts 56
6.1.2. Spring AOP capabilities and goals 57
6.1.3. AOP Proxies in Spring 58
6.2. Pointcuts in Spring 58
6.2.1. Concepts 59
6.2.2. Operations on pointcuts 59
6.2.3. Convenience pointcut implementations 60
6.2.4. Pointcut superclasses 61

6.2.5. Custom pointcuts ... 62
6.3. Advice types in Spring ... 62
6.3.1. Advice lifecycles ... 62
6.3.2. Advice types in Spring ... 62
6.4. Advisors in Spring ... 67
6.5. Using the ProxyFactoryBean to create AOP proxies ... 67
6.5.1. Basics ... 67
6.5.2. JavaBean properties ... 68
6.5.3. Proxying interfaces ... 69
6.5.4. Proxying classes ... 70
6.5.5. Using 'global' advisors ... 71
6.6. Convenient proxy creation ... 71
6.6.1. TransactionProxyFactoryBean ... 71
6.6.2. EJB proxies ... 72
6.7. Concise proxy definitions ... 73
6.8. Creating AOP proxies programmatically with the ProxyFactory ... 74
6.9. Manipulating advised objects ... 74
6.10. Using the "autoproxy" facility ... 75
6.10.1. Autoproxy bean definitions ... 76
6.10.2. Using metadata-driven auto-proxying ... 77
6.11. Using TargetSources ... 79
6.11.1. Hot swappable target sources ... 79
6.11.2. Pooling target sources ... 80
6.11.3. Prototype target sources ... 81
6.11.4. ThreadLocal target sources ... 81
6.12. Defining new Advice types ... 82
6.13. Further reading and resources ... 82
7. AspectJ Integration ...
7.1. Overview ... 83
7.2. Configuring AspectJ aspects using Spring IoC ... 83
7.2.1. "Singleton" aspects ... 83
7.2.2. Non-singleton aspects ... 84
7.2.3. Gotchas ... 84
7.3. Using AspectJ pointcuts to target Spring advice ... 84
7.4. Spring aspects for AspectJ ... 85
8. Transaction management ...
8.1. The Spring transaction abstraction ... 86
8.2. Transaction strategies ... 87
8.3. Resource synchronization with transactions ... 89
8.3.1. High-level approach ... 89
8.3.2. Low-level approach ... 89
8.3.3. TransactionAwareDataSourceProxy ... 90
8.4. Programmatic transaction management ... 90
8.4.1. Using the TransactionTemplate ... 91
8.4.2. Using the PlatformTransactionManager ... 91
8.5. Declarative transaction management ... 92
8.5.1. Source Annotations for Transaction Demarcation ... 94
8.5.2. BeanNameAutoProxyCreator, another declarative approach ... 97
8.5.3. AOP and Transactions ... 99
8.6. Choosing between programmatic and declarative transaction management ... 99

8.7. Do you need an application server for transaction management? 99
8.8. AppServer-specific integration 99
8.8.1. BEA WebLogic 100
8.8.2. IBM WebSphere 100
8.9. Common problems 100
8.9.1. Use of the wrong transaction manager for a specific DataSource 100
8.9.2. Spurious AppServer warnings about the transaction or DataSource no longer being active 100
9. Source Level Metadata Support .....
9.1. Source-level metadata 102
9.2. Spring's metadata support 103
9.3. Integration with Jakarta Commons Attributes 104
9.4. Metadata and Spring AOP autoproxying 105
9.4.1. Fundamentals 105
9.4.2. Declarative transaction management 106
9.4.3. Pooling 106
9.4.4. Custom metadata 107
9.5. Using attributes to minimize MVC web tier configuration 108
9.6. Other uses of metadata attributes 110
9.7. Adding support for additional metadata APIs 110
10. DAO support .....
10.1. Introduction 111
10.2. Consistent Exception Hierarchy 111
10.3. Consistent Abstract Classes for DAO Support 112
11. Data Access using JDBC .....
11.1. Introduction 113
11.2. Using the JDBC Core classes to control basic JDBC processing and error handling 113
11.2.1. JdbcTemplate 113
11.2.2. DataSource 114
11.2.3. SQLExceptionTranslator 114
11.2.4. Executing Statements 115
11.2.5. Running Queries 115
11.2.6. Updating the database 116
11.3. Controlling how we connect to the database 117
11.3.1. DataSourceUtils 117
11.3.2. SmartDataSource 117
11.3.3. AbstractDataSource 117
11.3.4. SingleConnectionDataSource 117
11.3.5. DriverManagerDataSource 117
11.3.6. TransactionAwareDataSourceProxy 118
11.3.7. DataSourceTransactionManager 118
11.4. Modeling JDBC operations as Java objects 118
11.4.1. SqlQuery 118
11.4.2. MappingSqlQuery 119
11.4.3. SqlUpdate 120
11.4.4. StoredProcedure 120
11.4.5. SqlFunction 121
12. Data Access using O/R Mappers .....
12.1. Introduction 122

12.2. Hibernate ... 123
12.2.1. Resource management ... 123
12.2.2. SessionFactory setup in a Spring application context ... 124
12.2.3. Inversion of Control: HibernateTemplate and HibernateCallback ... 125
12.2.4. Implementing Spring-based DAOs without callbacks ... 126
12.2.5. Implementing DAOs based on plain Hibernate3 API ... 126
12.2.6. Programmatic transaction demarcation ... 127
12.2.7. Declarative transaction demarcation ... 128
12.2.8. Transaction management strategies ... 130
12.2.9. Container resources versus local resources ... 131
12.2.10. Spurious AppServer warnings about the transaction or DataSource no longer being active ... 132
12.3. JDO ... 133
12.3.1. PersistenceManagerFactory setup ... 133
12.3.2. JdoTemplate and JdoDaoSupport ... 134
12.3.3. Implementing DAOs based on plain JDO API ... 135
12.3.4. Transaction management ... 137
12.3.5. JdoDialect ... 137
12.4. Oracle TopLink ... 138
12.4.1. SessionFactory abstraction ... 138
12.4.2. TopLinkTemplate and TopLinkDaoSupport ... 139
12.4.3. Implementing DAOs based on plain TopLink API ... 140
12.4.4. Transaction management ... 142
12.5. Apache OJB ... 142
12.5.1. OJB setup in a Spring environment ... 143
12.5.2. PersistenceBrokerTemplate and PersistenceBrokerDaoSupport ... 143
12.5.3. Transaction management ... 144
12.6. iBATIS SQL Maps ... 145
12.6.1. Overview and differences between iBATIS 1.x and 2.x ... 145
12.6.2. iBATIS SQL Maps 1.x ... 146
12.6.3. iBATIS SQL Maps 2.x ... 147
13. Web MVC framework ...
13.1. Introduction to the web MVC framework ... 151
13.1.1. Pluggability of other MVC implementations ... 151
13.1.2. Features of Spring MVC ... 152
13.2. The DispatcherServlet ... 152
13.3. Controllers ... 155
13.3.1. AbstractController and WebContentGenerator ... 155
13.3.2. Other simple controllers ... 156
13.3.3. The MultiActionController ... 157
13.3.4. CommandControllers ... 158
13.4. Handler mappings ... 159
13.4.1. BeanNameUrlHandlerMapping ... 160
13.4.2. SimpleUrlHandlerMapping ... 161
13.4.3. Adding HandlerInterceptors ... 161
13.5. Views and resolving them ... 163
13.5.1. ViewResolvers ... 163
13.5.2. Chaining ViewResolvers ... 164
13.5.3. Redirecting to views ... 165
13.6. Using locales ... 166
13.6.1. AcceptHeaderLocaleResolver ... 166

13.6.2. CookieLocaleResolver 167
13.6.3. SessionLocaleResolver 167
13.6.4. LocaleChangeInterceptor 167
13.7. Using themes 168
13.7.1. Introduction 168
13.7.2. Defining themes 168
13.7.3. Theme resolvers 168
13.8. Spring's multipart (fileupload) support 169
13.8.1. Introduction 169
13.8.2. Using the MultipartResolver 169
13.8.3. Handling a fileupload in a form 170
13.9. Handling exceptions 171
14. Integrating view technologies ....
14.1. Introduction 173
14.2. JSP & JSTL 173
14.2.1. View resolvers 173
14.2.2. 'Plain-old' JSPs versus JSTL 173
14.2.3. Additional tags facilitating development 174
14.3. Tiles 174
14.3.1. Dependencies 174
14.3.2. How to integrate Tiles 174
14.4. Velocity & FreeMarker 175
14.4.1. Dependencies 175
14.4.2. Context configuration 176
14.4.3. Creating templates 176
14.4.4. Advanced configuration 176
14.4.5. Bind support and form handling 177
14.5. XSLT 183
14.5.1. My First Words 183
14.5.2. Summary 185
14.6. Document views (PDF/Excel) 186
14.6.1. Introduction 186
14.6.2. Configuration and setup 186
14.7. JasperReports 188
14.7.1. Dependencies 188
14.7.2. Configuration 189
14.7.3. Populating the ModelAndView 191
14.7.4. Working with Sub-Reports 191
14.7.5. Configuring Exporter Parameters 192
15. Integrating with other web frameworks ....
15.1. Introduction 194
15.2. JavaServer Faces 195
15.2.1. DelegatingVariableResolver 195
15.2.2. FacesContextUtils 195
15.3. Struts 196
15.3.1. ContextLoaderPlugin 196
15.3.2. ActionSupport Classes 198
15.4. Tapestry 198
15.4.1. Architecture 199
15.4.2. Implementation 200
15.4.3. Summary 205
15.5. WebWork 205

16. **Remoting and web services using Spring**
16.1. Introduction ... 207
16.2. Exposing services using RMI ... 208
16.2.1. Exporting the service using the RmiServiceExporter ... 208
16.2.2. Linking in the service at the client ... 209
16.3. Using Hessian or Burlap to remotely call services via HTTP ... 209
16.3.1. Wiring up the DispatcherServlet for Hessian ... 209
16.3.2. Exposing your beans by using the HessianServiceExporter ... 210
16.3.3. Linking in the service on the client ... 210
16.3.4. Using Burlap ... 210
16.3.5. Applying HTTP basic authentication to a service exposed through Hessian or Burlap ... 210
16.4. Exposing services using HTTP invokers ... 211
16.4.1. Exposing the service object ... 211
16.4.2. Linking in the service at the client ... 211
16.5. Web Services ... 212
16.5.1. Exposing services using JAX-RPC ... 212
16.5.2. Accessing Web Services ... 213
16.5.3. Register Bean Mappings ... 214
16.5.4. Registering our own Handler ... 214
16.5.5. Exposing web services using XFire ... 215
16.6. Auto-detection is not implemented for remote interfaces ... 216
16.7. Considerations when choosing a technology ... 217
17. **Accessing and implementing EJBs**
17.1. Accessing EJBs ... 218
17.1.1. Concepts ... 218
17.1.2. Accessing local SLSBs ... 218
17.1.3. Accessing remote SLSBs ... 220
17.2. Using Spring convenience EJB implementation classes ... 220
18. **JMS**
18.1. Introduction ... 223
18.2. Domain unification ... 223
18.3. JmsTemplate ... 223
18.3.1. ConnectionFactory ... 224
18.3.2. Transaction Management ... 224
18.3.3. Destination Management ... 225
18.4. Using the JmsTemplate ... 225
18.4.1. Sending a message ... 226
18.4.2. Synchronous Receiving ... 226
18.4.3. Using Message Converters ... 227
18.4.4. SessionCallback and ProducerCallback ... 227
19. **JMX Support**
19.1. Introduction ... 229
19.2. Exporting your Beans to JMX ... 229
19.2.1. Creating an MBeanServer ... 230
19.2.2. Lazy-Initialized MBeans ... 231
19.2.3. Automatic Registration of MBeans ... 231
19.3. Controlling the Management Interface of Your Beans ... 231
19.3.1. The MBeanInfoAssembler Interface ... 231
19.3.2. Using Source-Level Metadata ... 231
19.3.3. Using JDK 5.0 Annotations ... 233
19.3.4. Source-Level Metadata Types ... 234

19.3.5. The AutodetectCapableMBeanInfoAssembler Interface 236
19.3.6. Defining Management Interfaces using Java Interfaces 236
19.3.7. Using MethodNameBasedMBeanInfoAssembler 238
19.4. Controlling the ObjectNames for your Beans 238
19.4.1. Reading ObjectNames from Properties 238
19.4.2. Using the MetadataNamingStrategy 239
19.5. Exporting your Beans with JSR-160 Connectors 239
19.5.1. Server-side Connectors 239
19.5.2. Client-side Connectors 240
19.5.3. JMX over Burlap/Hessian/SOAP 240
19.6. Accessing MBeans via Proxies 241
20. JCA CCI .....
20.1. Introduction 242
20.2. Configuring CCI 242
20.2.1. Connector configuration 242
20.2.2. ConnectionFactory configuration in Spring 243
20.2.3. Configuring CCI connections 243
20.2.4. Using a single CCI connection 244
20.3. Using Spring's CCI access support 244
20.3.1. Record conversion 244
20.3.2. CciTemplate 245
20.3.3. DAO support 247
20.3.4. Automatic output record generation 247
20.3.5. Summary 247
20.3.6. Using a CCI Connection and Interaction directly 248
20.3.7. Example for CciTemplate usage 249
20.4. Modeling CCI access as operation objects 250
20.4.1. MappingRecordOperation 251
20.4.2. MappingCommAreaOperation 251
20.4.3. Automatic output record generation 252
20.4.4. Summary 252
20.4.5. Example for MappingRecordOperation usage 252
20.4.6. Example for MappingCommAreaOperation usage 254
20.5. Transactions 255
21. Sending Email with Spring mail abstraction layer .....
21.1. Introduction 256
21.2. Spring mail abstraction structure 256
21.3. Using Spring mail abstraction 257
21.3.1. Pluggable MailSender implementations 259
21.4. Using the JavaMail MimeMessageHelper 259
21.4.1. Creating a simple MimeMessage and sending it 259
21.4.2. Sending attachments and inline resources 260
22. Scheduling jobs using Quartz or Timer .....
22.1. Introduction 261
22.2. Using the OpenSymphony Quartz Scheduler 261

22.2.1. Using the JobDetailBean 261
22.2.2. Using the MethodInvokingJobDetailFactoryBean 262
22.2.3. Wiring up jobs using triggers and the SchedulerFactoryBean 262
22.3. Using JDK Timer support 263
22.3.1. Creating custom timers 263
22.3.2. Using the MethodInvokingTimerTaskFactoryBean 264
22.3.3. Wrapping up: setting up the tasks using the TimerFactoryBean 264
23. Testing .....
23.1. Unit testing 266
23.2. Integration testing 266
23.2.1. Context management and caching 267
23.2.2. Dependency Injection of test class instances 267
23.2.3. Transaction management 267
23.2.4. Convenience variables 268
23.2.5. Example 268
23.2.6. Running integration tests 270
A. spring-beans.dtd .....

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 webservices, mailing facilities and various options in persisting your data to a database. Spring provides an MVC framework, transparent ways of integrating AOP into your software and a well-structured exception hierarchy including automatic mapping from proprietary exception hierarchies.

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

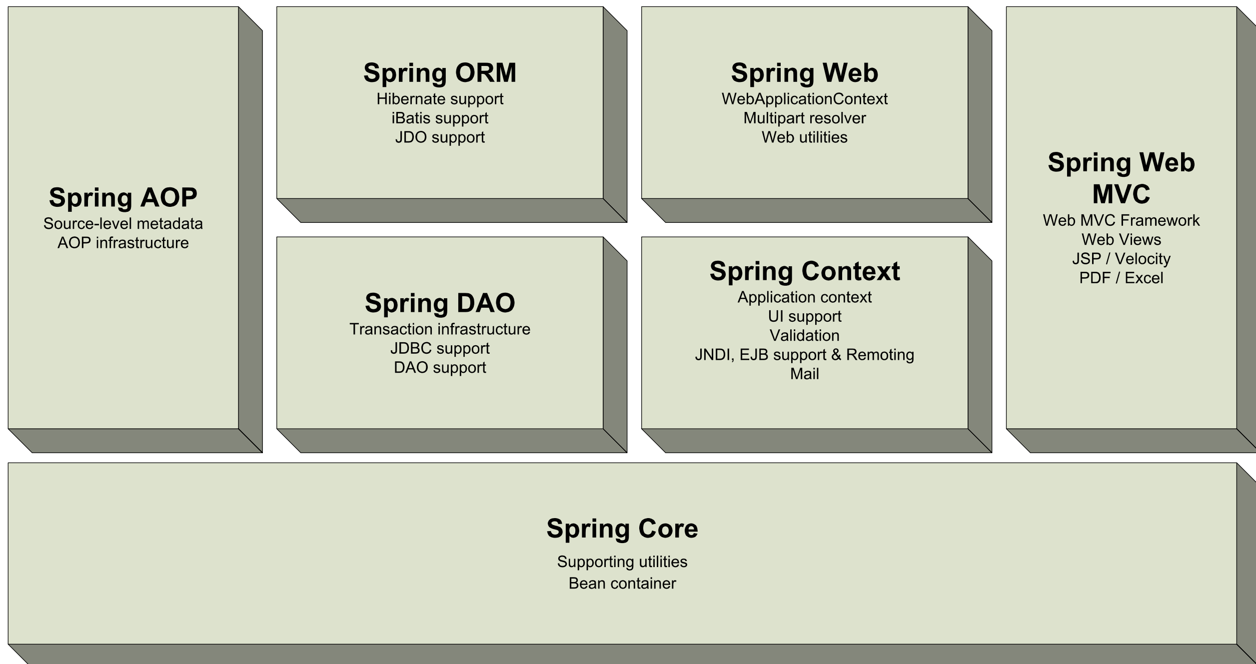
This document provides a reference guide to Spring's features. Since this document is still a work-in-progress, if you have any requests or comments, please post them on the user mailing list or on the forum at the SourceForge project page: http://www.sf.net/projects/springframework

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

Chapter 1. Introduction

1.1. Overview

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



Overview of the the Spring Framework

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

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

The *DAO* package provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing 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 JDO, Hibernate and iBatis. Using the ORM package you can use all those O/R-mappers in combination with all the other features Spring offers, like simple declarative transaction management mentioned before.

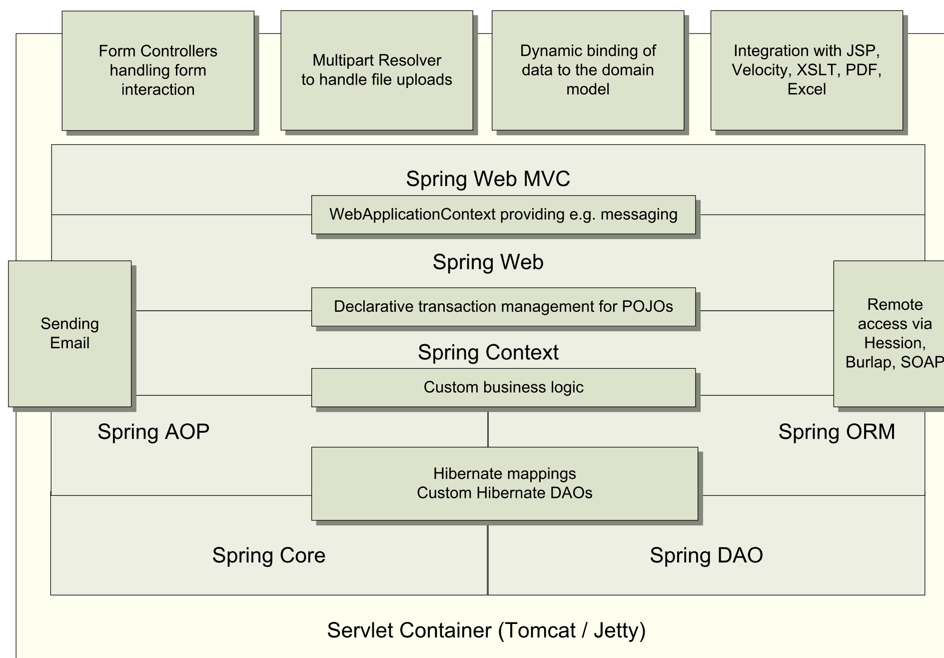
Spring's *AOP* package provides an *AOP Alliance* compliant aspect-oriented programming implementation allowing you to define, for example, method-interceptors and pointcuts to cleanly decouple code implementing functionality that should logically speaking be separated. Using source-level metadata functionality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

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

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

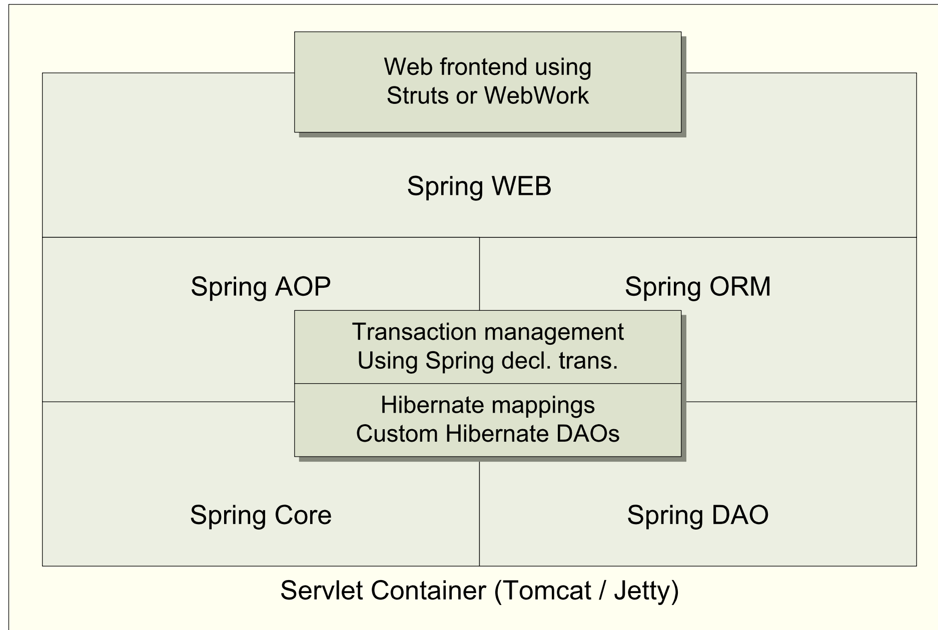
1.2. Usage scenarios

With the building 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.



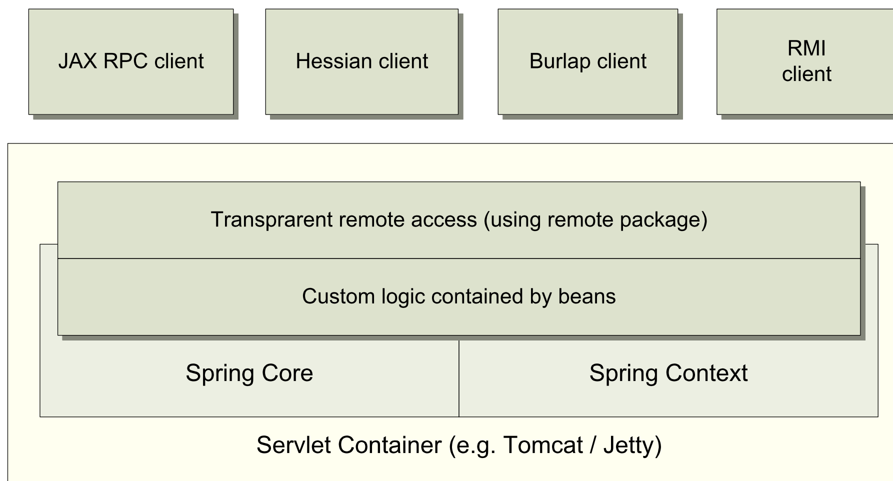
Typical full-fledged Spring web application

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



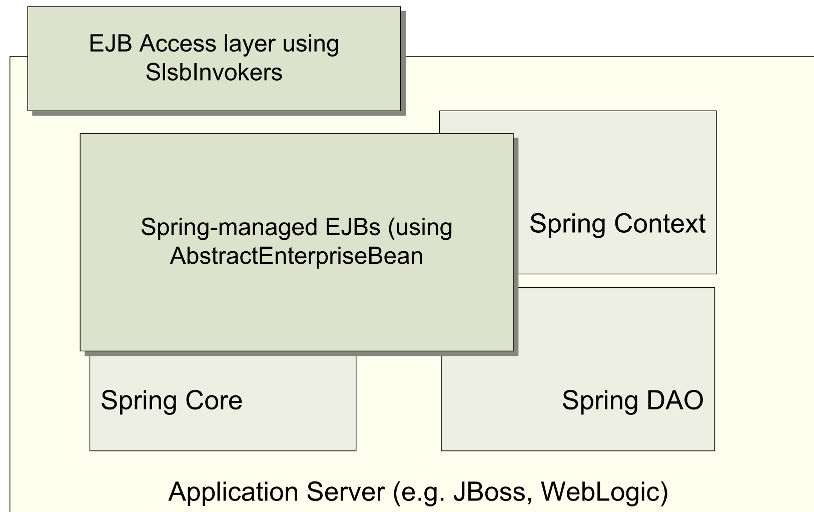
Spring middle-tier using a third-party web framework

Sometimes the current circumstances do not allow you to completely switch to a different framework. Spring does *not* force you to use everything within it; it's not an *all-or-nothing* solution. Existing frontends 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 UI layer using a `WebApplicationContext`.



Remoting usage scenario

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



EJBs - Wrapping existing POJOs

Spring also provides an access layer 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. Background information

2.1. Inversion of Control / Dependency Injection

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

Chapter 3. Beans, BeanFactory and the ApplicationContext

3.1. Introduction

Two of the most fundamental and important packages in Spring are the `org.springframework.beans` and `org.springframework.context` packages. Code in these packages provides the basis for Spring's *Inversion of Control* (alternately called *Dependency Injection*) features. The `BeanFactory` [http://www.springframework.org/docs/api/org/springframework/beans/factory/BeanFactory.html] provides an advanced configuration mechanism capable of managing beans (objects) of any nature, using potentially any kind of storage facility. The ApplicationContext [http://www.springframework.org/docs/api/org/springframework/context/ApplicationContext.html] builds on top of the BeanFactory (it's a subclass) and adds other functionality such as easier integration with Springs AOP features, message resource handling (for use in internationalization), event propagation, declarative mechanisms to create the ApplicationContext and optional parent contexts, and application-layer specific contexts such as the `WebApplicationContext`, among other enhancements.

In short, the `BeanFactory` provides the configuration framework and basic functionality, while the `ApplicationContext` adds enhanced capabilities to it, some of them perhaps more J2EE and enterprise-centric. In general, an ApplicationContext is a complete superset of a BeanFactory, and any description of BeanFactory capabilities and behavior should be considered to apply to ApplicationContexts as well.

Users are sometimes unsure whether a BeanFactory or an ApplicationContext are best suited for use in a particular situation. Normally when building most applications in a J2EE-environment, *the best option is to use the ApplicationContext*, since it offers all the features of the BeanFactory and adds on to it in terms of features, while also allowing a more declarative approach to use of some functionality, which is generally desirable. The main usage scenario when you might prefer to use the BeanFactory is when memory usage is the greatest concern (such as in an applet where every last kilobyte counts), and you don't need all the features of the ApplicationContext.

This chapter covers material related to both the BeanFactory and the ApplicationContext. When mention is made only of the BeanFactory, you may always assume the text also applies to the ApplicationContext. When functionality is only available in the ApplicationContext, explicit mention is made of this.

3.2. BeanFactory and BeanDefinitions - the basics

3.2.1. The BeanFactory

The `BeanFactory` [http://www.springframework.org/docs/api/org/springframework/beans/factory/BeanFactory.html] is the actual *container* which instantiates, configures, and manages a number of beans. These beans typically collaborate with one another, and thus have dependencies between themselves. These dependencies are reflected in the configuration data used by the BeanFactory (although some dependencies may not be visible as configuration data, but rather be a function of programmatic interactions between beans at runtime).

A BeanFactory is represented by the interface `org.springframework.beans.factory.BeanFactory`, for which there are multiple implementations. The most commonly used simple BeanFactory implementation is `org.springframework.beans.factory.xml.XmlBeanFactory`. (This should be qualified with the reminder that

ApplicationContexts are a subclass of BeanFactory, and most users end up using XML variants of ApplicationContext).

Although for most scenarios, almost all user code managed by the BeanFactory does not have to be aware of the BeanFactory, the BeanFactory does have to be instantiated somehow. This can happen via explicit user code such as:

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

or

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

or

```
ClassPathXmlApplicationContext appContext = new ClassPathXmlApplicationContext(
        new String[] {"applicationContext.xml", "applicationContext-part2.xml"});
// of course, an ApplicationContext is just a BeanFactory
BeanFactory factory = (BeanFactory) appContext;
```

Note: once you have learned the basics about bean factories and applicaiton contexts, from this chapter, it will also be useful to learn about Spring's `Resource` abstraction, as described in Chapter 4, *Abstracting Access to Low-Level Resources*. 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 (i.e. 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. Another special prefix, `classpath*:`, allows all context definiton files of the same name on the classpath to be found and combined to build a context. Please see the chapter referenced above for much more information on the topic of `Resource`s.

For many usage scenarios, user code will not have to instantiate the BeanFactory or ApplicationContext, since Spring Framework code will do it. For example, the web layer provides support code to load a Spring ApplicationContext automatically as part of the normal startup process of a J2EE web-app. This declarative process is described here:

While programmatic manipulation of BeanFactories will be described later, the following sections will concentrate on describing the configuration of BeanFactories.

A BeanFactory configuration consists of, at its most basic level, definitions of one or more beans that the BeanFactory must manage. In an XmlBeanFactory, these are configured as one or more `bean` elements inside a top-level `beans` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

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

  ...

</beans>
```

3.2.2. The BeanDefinition

Bean definitions inside a DefaultListableBeanFactory variant (like XmlBeanFactory) are represented as BeanDefinition objects, which contain (among other information) the following details:

- a class name: this is normally the actual implementation class of the bean being described in the bean definition. However, if the bean is to be constructed by calling 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 (i.e. prototype or singleton, autowiring mode, dependency checking mode, initialization and destruction methods)
- 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 a bean needs to do its work, i.e. *collaborators* (also specified as properties or as constructor arguments). These can also be called dependencies.

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

Table 3.1. Bean definition explanation

Feature	More info
class	Section 3.2.3, “The bean class”
id and name	Section 3.2.4, “The bean identifiers (id and name)”
singleton or prototype	Section 3.2.5, “To singleton or not to singleton”
constructor arguments	Section 3.3.1, “Setting bean properties and collaborators”
bean properties	Section 3.3.1, “Setting bean properties and collaborators”
autowiring mode	Section 3.3.6, “Autowiring collaborators”
dependency checking mode	Section 3.3.7, “Checking for dependencies”
initialization method	Section 3.4.1, “Lifecycle interfaces”
destruction method	Section 3.4.1, “Lifecycle interfaces”

Note that a bean definition is represented by the real interface `org.springframework.beans.factory.config.BeanDefinition`, and its various implementations (Root/ChildBeanDefinition). However, it is rare that user code works directly with BeanDefinition objects: Usually, bean definitions will be expressed in a metadata format (such as XML), which will be loaded on startup. The internal representation of such bean definitions are BeanDefinition objects in the factory.

Besides bean definitions which contain information on how to create a specific bean, a BeanFactory can also allow to register existing bean objects that have been created outside the factory (by custom code). DefaultListableBeanFactory supports this through the `registerSingleton` method, as defined by the `org.springframework.beans.factory.config.ConfigurableBeanFactory` interface. Typical applications solely work with beans defined through metadata bean definitions, though.

3.2.3. The bean class

The `class` attribute is normally mandatory (see Section 3.2.3.3, “Bean creation via instance factory method” and Section 3.5, “Abstract and child bean definitions” for the two exception) and is used for one of two purposes. In the much more common case where the BeanFactory itself directly creates the bean by calling its constructor (equivalent to Java code calling *new*), the class attribute specifies the class of the bean to be constructed. In the less common case where the BeanFactory calls a static, so-called *factory* method on a class to create the bean, the class attribute specifies the actual class containing the static factory method. (the type of the returned bean from the static factory method may be the same class or another class entirely, it doesn't matter).

3.2.3.1. Bean creation via constructor

When creating a bean using the constructor approach, all normal classes are usable by Spring 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 BeanFactory 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 BeanFactory, but it it's also possible to have more exotic non-bean-style classes in your BeanFactory. If, for example, you need to use a legacy connection pool that absolutely does not adhere to the JavaBean specification, no worries, Spring can manage it as well.

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

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

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

3.2.3.2. Bean creation via 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.

Following is an example of 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, `createInstance` 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.

3.2.3.3. Bean creation via instance factory method

Quite similar to using a static factory method to create a bean, is the use of an instance (non-static) factory method, where a factory method of an existing bean from the factory is called 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 an ancestor bean factory which contains the factory method. The factory method itself should still be set via the `factory-method` attribute.

Following is an example:

```
<!-- 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 Dependency Injection, by the container.

3.2.4. The bean identifiers (`id` and `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 BeanFactory or ApplicationContext 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.

In an XmlBeanFactory (including ApplicationContext variants), you use the `id` or `name` attributes to specify the bean id(s), and at least one id must be specified in one or both of these attributes. The `id` attribute allows you to specify one id, and as it is marked in the XML DTD (definition document) as a real XML element ID attribute, the parser is able to do some extra validation when other elements point back to this one. As such, it is the preferred way to specify a bean id. However, the XML spec does limit the characters which are legal in XML IDs. This is usually not really a constraint, but if you have a need to use one of these 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 (`,`) or semicolon (`;`) via the `name` attribute.

3.2.5. To singleton or not to singleton

Beans are defined to be deployed in one of two modes: singleton or non-singleton. (The latter is also called a prototype, although the term is used loosely as it doesn't quite fit). 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.

The non-singleton, prototype mode of a bean deployment results in the *creation of a new bean instance* every time a request for that specific bean is done. This is ideal for situations where for example each user needs an independent user object or something similar.

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

In the example below, two beans are declared of which one is defined as a singleton, and the other one is a non-singleton (prototype). `exampleBean` is created each and every time a client asks the BeanFactory for this

bean, while `yetAnotherExample` is only created once; a reference to the exact same instance is returned on each request for this bean.

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

Note: when deploying a bean in the prototype mode, the lifecycle of the bean changes slightly. By definition, Spring cannot manage the complete lifecycle of a non-singleton/prototype bean, since after it is created, it is given to the client and the container does not keep track of it at all any longer. You can think of Spring's role when talking about a non-singleton/prototype bean as a replacement for the 'new' operator. Any lifecycle aspects past that point have to be handled by the client. The lifecycle of a bean in the BeanFactory is further described in Section 3.4.1, “Lifecycle interfaces”.

3.3. Properties, collaborators, autowiring and dependency checking

3.3.1. Setting bean properties and collaborators

Inversion of Control has already been referred to as *Dependency Injection*. The basic principle is that beans define their dependencies (i.e. 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) of the bean instantiating or locating its dependencies on its own using direct construction of classes, or something like the *Service Locator* pattern. While we will not elaborate too much on the advantages of Dependency Injection, it becomes evident upon usage that code gets much cleaner 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 type they are.

As touched on in the previous paragraph, Inversion of Control/Dependency Injection exists in two major variants:

- *setter-based* dependency injection is realized by calling setters on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean. Beans defined in the BeanFactory that use setter-based dependency injection are *true JavaBeans*. Spring generally advocates usage of setter-based dependency injection, since a large number of constructor arguments can get unwieldy, especially when some properties are optional.
- *constructor-based* dependency injection is realized by invoking a constructor with a number of arguments, each representing a collaborator or property. 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. Although Spring generally advocates usage of setter-based dependency injection for most situations, it does fully support the constructor-based approach as well, since you may wish to use it with pre-existing beans which provide only multi-argument constructors, and no setters. Additionally, for simpler beans, some people prefer the constructor approach as a means of ensuring beans cannot be constructed in an invalid state.

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 JavaBeans `PropertyEditors` to know how to convert properties from one format to

another. The actual values being passed around are done in the form of `PropertyValue` objects. However, most users of Spring will not be dealing with these classes directly (i.e. programmatically), but rather with an XML definition file which will be converted internally into instances of these classes, and used to load an entire BeanFactory or ApplicationContext.

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 variant which 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-arg is either an actual definition of the value to set, or a reference to another bean in the BeanFactory. In the case of the ApplicationContext, the reference can be to a bean in a parent ApplicationContext.
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. Additionally, when talking about the XML based BeanFactory variants (including the ApplicationContext variants), these have built-in support for defining Lists, Maps, Sets, and Properties collection types. Additionally, Spring uses JavaBeans `PropertyEditor` definitions to be able to convert string values to other, arbitrary types. (You can provide the BeanFactory with your own `PropertyEditor` definitions to be able to convert your own custom types; more information about PropertyEditors and how to manually add custom ones, can be found in Section 3.9, “Registering additional custom PropertyEditors”). When a bean property is a Java Class type, Spring allows you to specify the value for that property as a string value which is the name of the class, and the `ClassEditor` PropertyEditor, which is built-in, will take care of converting that class name to an actual Class instance.
5. It is important to realize that Spring validates the configuration of each bean in the BeanFactory when the BeanFactory is created, including the validation that properties which are bean references are actually referring to valid beans (i.e. the beans being referred to are also defined in the BeanFactory, or in the case of ApplicationContext, a parent context). However, the bean properties themselves are not set until the bean *is actually created*. For beans which are singleton and set to be pre-instantiated (such as singleton beans in an ApplicationContext), creation happens at the time that the BeanFactory 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.
6. You can generally trust Spring to do the right thing. It will pick up configuration issues, including references to non-existent beans and circular dependencies, at BeanFactory load-time. It will actually set properties and resolve dependencies (i.e. create those dependencies if needed) as late as possible, which is when the bean is actually created. This does mean that a BeanFactory 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 by default pre-instantiates 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-load (not be pre-instantiated).

Some examples:

First, an example of using the BeanFactory for setter-based dependency injection. Below is a small part of an `XmlBeanFactory` configuration file specifying some bean definitions. Following is the code for the actual main bean itself, showing the appropriate setters declared.

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

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

```
public class ExampleBean {

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

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

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

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

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

Now, an example of using the BeanFactory for IoC type 3 (constructor-based dependency injection). Below is a snippet from an XML configuration that specifies constructor arguments and the actual bean code, showing the constructor:

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

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

```
public class ExampleBean {

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

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

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

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 bean="anotherExampleBean"/></constructor-arg>
  <constructor-arg><ref bean="yetAnotherBean"/></constructor-arg>
  <constructor-arg><value>1</value></constructor-arg>
</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. These arguments are optional. 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, mentioned previously, would be used in an essentially identical fashion (aside from the use of the `factory-bean` attribute instead of the `class` attribute), so will not be detailed here.

3.3.2. Constructor Argument Resolution

Constructor argument resolution matching occurs using the argument's type. When another bean is referenced, the type is known, and matching can occur. 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, which is used for the following two sections:

```
package examples;

public class ExampleBean {

    private int years;             //No. of years to the calculate the Ultimate Answer
    private String ultimateAnswer; //The Answer to Life, the Universe, and Everything

    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</value></constructor-arg>
  <constructor-arg type="java.lang.String"><value>42</value></constructor-arg>
</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</value></constructor-arg>
  <constructor-arg index="1"><value>42</value></constructor-arg>
</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*.

Specifying a constructor argument index is the preferred way of performing constructor IoC.

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. The `XmlBeanFactory` supports a number of sub-element types within its `property` and `constructor-arg` elements for this purpose.

3.3.3.1. The `value` element

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

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

3.3.3.2. The `null` element

The `null` element is used to handle null values. Spring treats empty arguments for properties and the like as empty Strings. The following XmlBeanFactory configuration:

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

results in the email property being set to "", equivalent to the java code: `exampleBean.setEmail("")`. The special `<null>` element may be used to indicate a null value, so that:

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

is equivalent to the java code: `exampleBean.setEmail(null)`.

3.3.3.3. The collection elements

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

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

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

3.3.3.4. Inner bean definitions via nested bean elements

A `bean` element inside the `property` element is used to define a bean value inline, instead of referring to a bean defined elsewhere in the BeanFactory. The inline bean definition does not need to have any id defined.

```
<bean id="outer" class="...">
  <!-- Instead of using a reference to target, just use an inner bean -->
  <property name="target">
    <bean class="com.mycompany.PersonImpl">
      <property name="name"><value>Tony</value></property>
      <property name="age"><value>51</value></property>
    </bean>
  </property>
</bean>
```

Note that the `singleton` flag and any `id` attribute are effectively ignored. Inner beans are anonymous prototypes.

3.3.3.5. The `idref` element

An idref element is simply a shorthand and error-proof way to set a property to the String *id* or *name* of another bean in the container.

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

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

This is exactly equivalent at runtime to the following fragment:

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

<bean id="theClientBean" 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 will allow Spring to validate at deployment time that the other bean actually exists. In the second variation, the class who's *targetName* property is forced to do its own validation, and that will only happen when that class is actually instantiated by Spring, possibly long after the container is actually deployed.

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

```
  <property name="targetName">
    <idref local="theTargetBean"/>
  </property>
```

3.3.3.6. The `ref` element

The `ref` element is the final element allowed inside a `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, so to speak. 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 BeanFactory/ApplicationContext (whether or not in the same XML file), or parent BeanFactory/ApplicationContext. 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 are 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 BeanFactory (or ApplicationContext) of the current BeanFactory (or ApplicationContext). 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 BeanFactory or ApplicationContext to the current one. The main use of this bean reference variant is when there is a need to wrap an existing bean in a parent context with some sort of proxy (which may have the same name as the parent), and needs the original object so it may wrap it.

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

3.3.3.7. Value and Ref shortcut forms

It is so common to need to configure a value or a bean reference, that there exist some shortcut forms which are less verbose than using the full `value` and `ref` elements. 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"/>
```

In general, when typing definitions by hand, you will probably prefer to use the less verbose shortcut form.

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 ref, 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">`.

3.3.3.8. Compound property names

Note that 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 non-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 a scalar value of 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 null-pointer exception will be thrown.

3.3.4. Method Injection

For most users, the majority of the beans in the container will be singletons. When a singleton bean needs to collaborate with (use) 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 however 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 its 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 problem is to forgo some inversion of control. Bean A can be aware of the container (as described here) by implementing `BeanFactoryAware`, and use programmatic means (as described here) to ask the container via a `getBean("B")` call for (a new) bean B every time it needs it. This is generally not a desirable solution since the bean code is then aware of and coupled to Spring.

Method Injection, an advanced feature of the BeanFactory, allows this use case to be handled in a clean fashion, along with some other scenarios.

3.3.4.1. Lookup method Injection

Lookup method injection refers to the ability of the container to override abstract or concrete methods on managed beans in the container, to return the result of looking up another named bean in the container. The lookup will typically be of a non-singleton bean as per the scenario described above (although it can also be a singleton). Spring implements this through a dynamically generated subclass overriding the method, using bytecode generation via the CGLIB library.

In the client class containing the method to be injected, the method definition must be an abstract (or concrete) definition in this form:

```
protected abstract SingleShotHelper createSingleShotHelper();
```

If the method is not abstract, Spring will simply override the existing implementation. In the XmlBeanFactory case, you instruct Spring to inject/override this method to return a particular bean from the container, by using the `lookup-method` element inside the bean definition. For example:

```
<!-- a stateful bean deployed as a prototype (non-singleton) -->
<bean id="singleShotHelper class="..." singleton="false">
</bean>

<!-- myBean uses singleShotHelper -->
<bean id="myBean" class="...">
  <lookup-method name="createSingleShotHelper" bean="singleShotHelper"/>
  <property>
    ...
  </property>
</bean>
```

The bean identified as *myBean* will call its own method `createSingleShotHelper` whenever it needs a new instance of the *singleShotHelper* bean. It is important to note that the person deploying the beans must be careful to deploy *singleShotHelper* as a non-singleton (if that is actually what is needed). If it is deployed as a singleton (either explicitly, or relying on the default *true* setting for this flag), the same instance of singleShotHelper will be returned each time!

Note that lookup method injection can be combined with Constructor Injection (supplying optional constructor arguments to the bean being constructed), and also with Setter Injection (settings properties on the bean being constructed).

3.3.4.2. 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.

In an XmlBeanFactory, 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 is needed to provide the new method definition.

```
/** meant to be used to override the existing computeValue
    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 BeanFactory deployment definition to deploy the original class and specify the method override would look like:

```
<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 just using the shortest string which will match an argument.

3.3.5. Using `depends-on`

For most situations, the fact that a bean is a dependency of another is expressed simply by the fact that one bean is set as a property of another. This is typically done with the `ref` element in the XmlBeanFactory. In a variation of this, sometimes a bean which is aware of the container is simply given the id of its dependency (using a string value or alternately the `idref` element, which evaluates the same as a string value). The first bean then programmatically asks the container for its dependency. In either case, the dependency is properly initialized before the dependent bean.

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` element may be used to explicitly force one or more beans to be initialized before the bean using this element is initialized.

Following is an example configuration:

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

```
  <property name="manager"><ref local="manager"/></property>
</bean>

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

3.3.6. Autowiring collaborators

A BeanFactory is able to *autowire* relationships between collaborating beans. This means it's 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 won't be autowired. Using autowiring, it is possible to reduce or eliminate the need to specify properties or constructor arguments, saving a significant amount of typing.[1]In an XmlBeanFactory, 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	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 BeanFactory 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 BeanFactory. 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 bean factory, a fatal error is raised.
autodetect	Chooses *constructor* or *byType* through introspection of the bean class. If a default constructor is found, byType gets applied.

Note that explicit dependencies in `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.

It's important to understand the pros and cons around autowiring. Some advantages of autowiring:

- It can significantly reduce the volume of configuration required. (However, mechanisms such as the use of a configuration "template," discussed elsewhere in this chapter, are also valuable here.)

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

- It 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:

- It's 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 is no longer explicitly documented.
- Wiring information may not be available to tools that may generate documentation from a Spring application context.
- 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. We recommend a degree of consistency across a project. For example, if autowiring is not used in general, it might be confusing to developers to use it just to one or two bean definitions.

3.3.7. Checking for dependencies

Spring has the ability to try to check for the existence of unresolved dependencies of a bean deployed into the BeanFactory. 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. In an XmlBeanFactory, 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, i.e. other beans)
object	Dependency checking is performed for collaborators
all	Dependency checking is done for collaborators, primitive types and collections

3.4. Customizing the nature of a bean

3.4.1. Lifecycle interfaces

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

Internally, Spring uses `BeanPostProcessors` to process any marker interfaces it can find and call the appropriate methods. If you need custom features or other lifecycle behavior Spring doesn't offer out-of-the-box, you can implement a `BeanPostProcessor` yourself. More information about this can be found in Section 3.7, “Customizing beans with BeanPostProcessors”.

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.

3.4.1.1. InitializingBean / `init-method`

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

```
    * Invoked by a BeanFactory after it has set all bean properties supplied
    * (and satisfied BeanFactoryAware and ApplicationContextAware).
    * <p>This method allows the bean instance to perform initialization only
    * possible when all bean properties have been set and to throw an
    * exception in the event of misconfiguration.
    * @throws Exception in the event of misconfiguration (such
    * as failure to set an essential property) or if initialization fails.
    */
    void afterPropertiesSet() throws Exception;
```

Note: generally, the use of the `InitializingBean` marker 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 the XmlBeanFactory, this is done via 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.

3.4.1.2. DisposableBean / `destroy-method`

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

```
    /**
    * Invoked by a BeanFactory on destruction of a singleton.
    * @throws Exception in case of shutdown errors.
    * Exceptions will get logged but not re-thrown to allow
    * other beans to release their resources too.
    */
    void destroy() throws Exception;
```

Note: 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. In the case of the XmlBeanFactory, this is done via the `destroy-method` attribute. 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 closing connection)
    }
}
```

Is exactly the same as:

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

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

but does not couple the code to Spring.

Important note: when deploying a bean in the prototype mode, the lifecycle of the bean changes slightly. By definition, Spring cannot manage the complete lifecycle of a non-singleton/prototype bean, since after it is created, it is given to the client and the container does not keep track of it at all any longer. You can think of Spring's role when talking about a non-singleton/prototype bean as a replacement for the 'new' operator. Any lifecycle aspects past that point have to be handled by the client. The lifecycle of a bean in the BeanFactory is further described in Section 3.4.1, "Lifecycle interfaces".

3.4.2. Knowing who you are

3.4.2.1. 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 {
   /**
    * Callback that supplies the owning factory to a bean instance.
    * <p>Invoked after population of normal bean properties but before an init
    * callback like InitializingBean's afterPropertiesSet or a custom init-method.
    * @param beanFactory owning BeanFactory (may not be null).
    * The bean can immediately call methods on the factory.
    * @throws BeansException in case of initialization errors
    * @see BeanInitializationException
    */
    void setBeanFactory(BeanFactory beanFactory) throws BeansException;
}
```

This allows beans to manipulate the BeanFactory that created them programmatically, through the `org.springframework.beans.factory.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.

3.4.2.2. 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 init callback like `InitializingBean`'s *afterPropertiesSet* or a custom init-method.

3.4.3. FactoryBean

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

3.5. Abstract and child bean definitions

A bean definition potentially contains a large amount of configuration information, including container specific information (i.e. initialization method, static factory method name, etc.) and constructor arguments and property values. A child bean definition is a bean definition which 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. In an XmlBeanFactory bean definition, 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"/>
  <!-- age should inherit value of 1 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, i.e. 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 init method, destroy method and/or static 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*, *singleton*, *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:

```
<bean id="inheritedTestBeanWithoutClass">
    <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 should inherit value of 1 from parent -->
</bean>
```

the parent bean cannot get instantiated on its own since it is incomplete, and it's also considered abstract. When a definition is considered abstract like this (explicitly or implicitly), it's usable just as a pure template or abstract bean definition that will serve as 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 considered abstract.

Important Note: Application contexts (but not simple bean factories) 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 pre-instantiate it.

3.6. Interacting with the BeanFactory

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");
XmlBeanFactory factory = new XmlBeanFactory(is);
```

Basically that's all there is to it. Using `getBean(String)` you can retrieve instances of your beans. You'll get a reference to the same bean if you defined it as a singleton (the default) or you'll get a new instance each time if you set `singleton` to *false*. The client-side view of the BeanFactory is surprisingly simple. The `BeanFactory` interface has only five methods for clients 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)
- `boolean isSingleton(String)`: determines whether or not the bean definition or bean instance registered under the given name is a singleton or a prototype. If no bean corresponding to the given name could not be found, an exception will be thrown (`NoSuchBeanDefinitionException`)
- `String[] getAliases(String)`: Return the aliases for the given bean name, if any were defined in the bean definition

3.6.1. Obtaining a FactoryBean, not its product

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

3.7. Customizing beans with BeanPostProcessors

A bean post-processor is a java class which implements the `org.springframework.beans.factory.config.BeanPostProcessor` interface, which consists of two callback methods. When such a class is registered as a post-processor with the BeanFactory, for each bean instance that is created by the BeanFactory, the post-processor will get a callback from the BeanFactory before any initialization methods (*afterPropertiesSet* and any declared init method) are called, and also afterwords. The post-processor is free to do what it wishes with the bean, 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 Spring helper classes are implemented as bean post-processors.

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 deployed into it which implement the `BeanPostProcessor` interface, and register them as post-processors, to be then called appropriately by the factory on bean creation. Nothing else needs to be done other than deploying the post-processor in a similar fashion to any other bean. On the other hand, when using plain BeanFactories, bean post-processors have to manually be *explicitly* registered, with a code sequence such as the following:

```
ConfigurableBeanFactory bf = new .....;     // create BeanFactory
   ...                        // now register some beans
// now register any needed BeanPostProcessors
MyBeanPostProcessor pp = new MyBeanPostProcessor();
bf.addBeanPostProcessor(pp);

// now start using the factory
  ...
```

Since this manual registration step is not convenient, and ApplictionContexts are functionally supersets of BeanFactories, it is generally recommended that ApplicationContext variants are used when bean post-processors are needed.

3.8. Customizing bean factories with BeanFactoryPostProcessors

A bean factory post-processor is a java class which implements the `org.springframework.beans.factory.config.BeanFactoryPostProcessor` interface. It is executed manually (in the case of the BeanFactory) or automatically (in the case of the ApplicationContext) to apply changes of some sort to an entire BeanFactory, after it has been constructed. Spring includes a number of pre-existing bean factory post-processors, such as `PropertyResourceConfigurer` and `PropertyPlaceHolderConfigurer`, both described below, and `BeanNameAutoProxyCreator`, 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 editors (as also mentioned in Section 3.9, “Registering additional custom PropertyEditors”).

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"));
// create placeholderconfigurer to 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);
```

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.

Since this manual step is not convenient, and ApplictionContexts are functionally supersets of BeanFactories, it is generally recommended that ApplicationContext variants are used when bean factory post-processors are needed.

3.8.1. The `PropertyPlaceholderConfigurer`

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

Consider a fragment from a BeanFactory definition, where a DataSource with placeholder values is defined:

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

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

The actual values come from another file in Properties format:

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

To use this with a BeanFactory, the bean factory post-processor is manually executed on it:

```
XmlBeanFactory factory = new XmlBeanFactory(new FileSystemResource("beans.xml"));
PropertyPlaceholderConfigurer cfg = new PropertyPlaceholderConfigurer();
cfg.setLocation(new FileSystemResource("jdbc.properties"));
cfg.postProcessBeanFactory(factory);
```

Note that ApplicationContexts are able to automatically recognize and apply beans deployed in them which implement BeanFactoryPostProcessor. This means that as described here, applying PropertyPlaceholderConfiguer is much more convenient when using an ApplicationContext. For this reason, it is recommended that users wishing to use this or other bean factory postprocessors use an ApplicationContext instead of a BeanFactory.

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 PropertiesPlaceholderConfigurer for more information.

3.8.2. 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 PropertyOverrideConfigurers 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 could look like:

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

This example file would be usable against a BeanFactory definition which contains a bean in it 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 overriden 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.

3.9. Registering additional custom PropertyEditors

When setting bean properties as a string value, a BeanFactory 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 to be simply 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 BeanFactories, this one has a nested property setup, so it is strongly recommended that, as described here, 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 next chapter. Additionally, ApplicationContexts also override or add an additional 3 editors to handle resource lookups in a manner appropriate to the specific application context type. Thee are: `InputStreamEditor`, `ResourceEditor` and `URLEditor`.

3.10. Using the alias element to add aliases for existing 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 spcified via the `id` attribute, and any number of other names via the `alias` 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 itslef.

Having to specify all alias when the bean is actually defined is not always adequate however. It is sometimes desirable to introduce an alias for a bean which is define elsewhere. This may be done via a standalone `alias` element.

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

In this case, a bean in the same context 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 fragement 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.

3.11. Introduction to the `ApplicationContext`

While the `beans` package provides basic functionality for managing and manipulating beans, often in a programmatic way, the `context` package adds `ApplicationContext` [http://www.springframework.org/docs/api/org/springframework/context/ApplicationContext.html], 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:

- *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 described functionality which ApplicationContext adds to basic BeanFactory capabilities.

3.12. Added functionality of the `ApplicationContext`

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

3.12.1. Using the `MessageSource`

The ApplicationContext interface extends an interface called `MessageSource`, and therefore provides messaging (i18n or internationalization) functionality. Together with the `NestingMessageSource`, 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 `NestingMessageSource` 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 provides an example for:

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

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

3.12.2. Propagating 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 3.4. Built-in Events

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

and then, the actual beans:

```
public class EmailBean implements ApplicationContextAware {

    /** the blacklist */
    private List blackList;

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

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

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

public class BlackListNotifier implement ApplicationListener {

    /** notification address */
    private String notificationAddress;

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

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

Of course, this 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.

3.12.3. Low-level resources and the application context

For optimal usage and understanding of application contexts, users should generally familiarize themselves with Spring's `Resource` abstraction, as described in Chapter 4, *Abstracting Access to Low-Level Resources*.

An application context is a `ResourceLoader`, able to be used to load `Resource`s. 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 pased 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 registerd 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 (i.e. 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.

The previously mentioned chapter provides much more information on these topics.

3.13. Customized behavior in the ApplicationContext

The BeanFactory already offers a number of mechanisms to control the lifecycle of beans deployed in it (such as marker interfaces like `InitializingBean` or `DisposableBean`, their configuration only equivalents such as the `init-method` and `destroy-method` attributes in an XmlBeanFactory config, and bean post-processors. In an ApplicationContext, all of these still work, but additional mechanisms are added for customizing behavior of beans and the container.

3.13.1. `ApplicationContextAware` marker interface

All marker interfaces available with BeanFactories still work. The ApplicationContext does add one extra marker interface which beans may implement, `org.springframework.context.ApplicationContextAware`. A bean which implements this interface and is deployed into the context will be called back on creation of the bean, using the interface's `setApplicationContext()` method, and provided with a reference to the context, which may be stored for later interaction with the context.

3.13.2. The `BeanPostProcessor`

Bean post-processors, java classes which implement the `org.springframework.beans.factory.config.BeanPostProcessor` interface, have already been mentioned. It is worth mentioning again here though, that post-processors are much more convenient to use in ApplicationContexts than in plain BeanFactories. In an ApplicationContext, any deployed bean which implements the above marker interface is automatically detected and registered as a bean post-processor, to be called appropriately at creation time for each bean in the factory.

3.13.3. The `BeanFactoryPostProcessor`

Bean factory post-processors, java classes which implement the

`org.springframework.beans.factory.config.BeanFactoryPostProcessor` interface, have already been mentioned. It is worth mentioning again here though, that bean factory post-processors are much more convenient to use in ApplicationContexts than in plain BeanFactories. In an ApplicationContext, any deployed bean which implements the above marker interface is automatically detected as a bean factory post-processor, to be called at the appropriate time.

3.13.4. The `PropertyPlaceholderConfigurer`

The `PropertyPlaceholderConfigurer` has already been described, as used with a BeanFactory. It is worth mentioning here though, that it is generally more convenient to use it with an ApplicationContext, since the context will automatically recognize and apply any bean factory post-processors, such as this one, when they are simply deployed into it like any other bean. There is no need for a manual step to execute it.

```
<!-- property placeholder post-processor -->
<bean id="placeholderConfig"
      class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="jdbc.properties"/>
</bean>
```

3.14. Registering additional custom PropertyEditors

As previously mentioned, standard JavaBeans PropertyEditors 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 PropertyEditors to an ApplicationContext.

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

```
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</value></property>
</bean>
```

The PropertyEditor could look similar to this:

```
// converts string representation to ExoticType object
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>
```

3.15. Setting a bean property or constructor arg from a property expression

`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" singleton="false">
  <property name="age"><value>10</value></property>
  <property name="spouse">
    <bean class="org.springframework.beans.TestBean">
      <property name="age"><value>11</value></property>
    </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</value></property>
  <property name="propertyPath"><value>spouse.age</value></property>
</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</value></property>
    </bean>
  </property>
  <property name="propertyPath"><value>age</value></property>
</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 curse, 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.

3.16. 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.

Here's 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</value></property>
</bean>
```

There's also a convenience usage form where the static field is specified as a bean name:

```
<bean id="java.sql.Connection.TRANSACTION_SERIALIZABLE"
      class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean"/>
```

This means 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's also possible to access a non-static field of another bean, as described in the JavaDocs.

3.17. Invoking another method and optionally using the return value.

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

We would however recommend that for the second use case, factory-methods, described previously, are a better all around choice.

An example (in an XML based BeanFactory definition) of a bean definition which uses this class to force some sort of static initialization:

```
<bean id="force-init" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="staticMethod"><value>com.example.MyClass.initialize</value></property>
</bean>

<bean id="bean1" class="..." depends-on="force-init">
  ...
</bean>
```

Note that the definition for `bean1` has used the `depends-on` attribute to refer to the `force-init` bean, which will trigger initializing `force-init` first, and thus calling the static initializer method, when `bean1` is first initialized.

Here's an example of a bean definition which uses this class to call a static factory method:

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

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

```
<bean id="sysProps" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetClass"><value>java.lang.System</value></property>
  <property name="targetMethod"><value>getProperties</value></property>
</bean>

<bean id="javaVersion" class="org.springframework.beans.factory.config.MethodInvokingFactoryBean">
  <property name="targetObject"><ref local="sysProps"/></property>
  <property name="targetMethod"><value>getProperty</value></property>
  <property name="arguments">
    <list>
      <value>java.version</value>
    </list>
  </property>
</bean>
```

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

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

3.18. Importing Bean Definitions from One File Into Another

It's often 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 configurations files unaware of the fact that they are being combined with others. However, an alternate approach is to from one XML bean definition file, use one or more instances of the `import` element to load definitions from one or more other files. Any `import` elements must be placed before `bean` elements in the file doing the importing. 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="..."/>
  . . .
```

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 DTD, including the top level `beans` element.

3.19. Creating an ApplicationContext from a web application

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

The ContextLoader has two implementations: the `ContextLoaderListener` and the `ContextLoaderServlet`. They both have the same functionality but differ in that the listener cannot be used in Servlet 2.2 compatible containers. Since the Servlet 2.4 specification, listeners are required to initialize after startup of a web application. A lot of 2.3 compatible containers already implement this feature. It is up to you as to which one you use, but all things being equal you should probably prefer `ContextLoaderListener`; for more information on compatibility, have a look at the JavaDoc for the `ContextLoaderServlet`.

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

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

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

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

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

3.20. Glue code and the evil singleton

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

As another example, in a complex J2EE apps with multiple layers (i.e. various JAR files, EJBs, and WAR files packaged as an EAR), with each layer having its own ApplicationContext definition (effectively forming a hierarchy), the preferred approach when there is only one web-app (WAR) in the top hierarchy is to simply create one composite ApplicationContext from the multiple XML definition files from each layer. All the ApplicationContext variants may be constructed from multiple definition files in this fashion. However, if there are multiple sibling web-apps at the top of the hierarchy, it is problematic to create an ApplicationContext for each web-app which 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 (i.e. a Hibernate SessionFactory), and possible issues due to side-effects. As an alternative, classes such as `ContextSingletonBeanFactoryLocator` [???] or `SingletonBeanFactoryLocator` [http://www.springframework.org/docs/api/org/springframework/beans/factory/access/SingletonBeanFactoryL may be used to demand load multiple hierarchical (i.e. one is a parent of another) BeanFactories or ApplicationContexts in an effectively singleton fashion, which may then be used as the parents of the web-app ApplicationContexts. The result is that bean definitions for lower layers are loaded only as needed, and loaded only once.

3.20.1. Using SingletonBeanFactoryLocator and ContextSingletonBeanFactoryLocator

You can see a detailed example of using `SingletonBeanFactoryLocator`

[http://www.springframework.org/docs/api/org/springframework/beans/factory/access/SingletonBeanFactoryL
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. Abstracting Access to Low-Level Resources

4.1. Overview

Java's standard `java.net.URL` interface and istandard handlers for vairous URL prefixes are unfortunately not quite adequate enough for all access to low-level resources. There is for example no standardized `URL` implementation which may be used to access a resource that needs to be obtained from somewhere on the classpath, or relative to a `ServletContext`, for example. 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 desireable functionality, such as a method to check 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 are:

- `getInputStream()`: locates and opens the resource, returning an `InputStream` for reading it. 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

Other methods allow you to obtain an actual URL or File object representing the resource, if the underlaying implementation is compatible, and supports that functionality.

`Resource` 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 should be created and used. Internally, a JavaBeans `PropertyEditor` is used to convert the `String` to the appropriate `Resource` type, but this is irrelevant to the user.

While `Resource` 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's important to note that Resource doesn't replace functionality, it wraps it where possible. For example, a UrlResource wraps a URL, and uses the wrapped URL to do its work.

4.3. Built-in Resource implementations

There are a number of built-in Resource implementations.

4.3.1. `UrlResource`

This 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 vs. 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 prefiix, 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 implementation of `Resource` 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. It always supports resolution as `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 web application 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 `ByteArrayInputStreams` 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` Interface

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

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

All application contexts implement `ResourceLoader` therefore all application contexts may be used to obtain `Resources`.

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, if you ask a ClassPathXmlApplicationContext

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

you'll get back a `ClassPathResource`, but if the same method is called on a FileSystemXmlApplicationContext, you'd get back a FileSystemResource. For a `WebApplicationContext`, you'd get 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);
```

or force a UrlResource to be used by specifyng 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);
```

4.5. The `ResourceLoaderAware` interface

The `ResourceLoaderAware` interface is a special marker interface, for objects that expect to be provided with a ResourceLoader:

```
public interface ResourceLoaderAware {
   void setResourceLoader(ResourceLoader resourceLoader);
}
```

When a bean implements `ResourceLoaderAware` and is deployed into an application context, it is recognized by the application context and called back by it, with the application context itself passed in as the `ResourceLoader` argument.

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

4.6. Setting `Resource`s as properties

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 one needed that depends on the role of the user. If on the other hand 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 text 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`, `ServletContextResource`, etc., as appropriate depending on the type of the context.

If there is a need to force a specifc `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 constuctor (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 definiton, 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");
```

then the definition will be loaded from the classpath, as a ClassPathResource will be used. But if you create a FilleSystemXmlApplicationContext as follows:

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

then the 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 its definition from the classpath. However, it's still a `FileSystemXmlApplicationContext`. If it's subsequently used as a `ResourceLoader`, any unprefixed paths are still treated as filesystem paths.

4.7.2. 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 gven name should be obtained (internally, this essentially happens via a `ClassLoader.getResources(...)` call), and then merged to form the final application context definition.

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 special prefix is specific to application contexts, 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.

4.7.3. Unexpected application context handling of FileSystemResource absolute paths

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. FileSystemApplicationContext` simply forces all attached `FileSystemResources` 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 FileSystemClassPathXmlApplicationContext("conf/context.xml");
```

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

as well as the following

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

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

Even though it would make sense for them to be different, as one case being relative vs. one being absolute.

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. PropertyEditors, data binding, validation and the BeanWrapper

5.1. Introduction

The big question is whether or not validation should be considered *business logic*. There are pros and cons for both answers, and Spring offers a design for validation (and data binding) that does not exclude either one of them. Validation should specifically not be tied to the web tier, should be 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's both basic and usable in every layer of an application.

Data binding is useful for allowing user input to be dynamically bound to the domain model of an application (or whatever objects you use to process user input). Spring provides the so-called `DataBinder` to do exactly that. The Validator and the DataBinder make up the `validation` package, which is primarily used in but not limited to the MVC framework.

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

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

5.2. Binding data using the `DataBinder`

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

5.3. Bean manipulation and the `BeanWrapper`

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

One quite important concept of the beans package is the `BeanWrapper` interface and its corresponding implementation (`BeanWrapperImpl`). As quoted from the JavaDoc, the BeanWrapper offers 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`

[2]See the beans chapter for more information

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.

5.3.1. Setting and getting basic and nested properties

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

Table 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.

Note: this part is not important to you if you're not planning to work with the BeanWrapper directly. If you're just using the `DataBinder` and the `BeanFactory` and their out-of-the-box implementation, 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 float salary;

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

```
        this.salary = salary;
    }
}
```

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

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

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

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

5.3.2. Built-in `PropertyEditors`, converting types

Spring heavily uses the concept of `PropertyEditors`. Sometimes it might be handy to be able to represent properties in a different way than the object itself. For example, a date can be represented in a human readable way, while we're still able to convert the human readable form back to the original date (or even better: convert any date entered in a human readable form, back to Date objects). This behavior can be achieved by *registering custom editors*, of type `java.beans.PropertyEditor`. Registering custom editors on a BeanWrapper or alternately in a specific Application Context 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

Class	Explanation
	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.
`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 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.
`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.
`StringArrayPropertyEditor`	Capable of resolving a comma-delimited list of String to a String-array and vice versa. 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 PropertyEditors for Font, Color and all the primitive types. Note also that the standard JavaBeans infrastructure will automatically discover PropertyEditors (without you having to register them) if they are in the same package as the class they handle, and have the same name as that class, with 'Editor' appended.

5.3.3. Other features worth mentioning

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

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

5.4. Validation using Spring's Validator interface

Spring's features a Validator interface you can use to validate objects. The Validator interface, is pretty straightforward and works using with a so-called Errors object. In other words, while validating, validators will report validation failures to the Errors object.

As said already, the Validator interface is pretty straightforward, just as implementing one yourself. Let's consider a small data object:

```
public class Person {
  private String name;
  private int age;

  // the usual suspects: getters and setters
}
```

Using the `org.springframework.validation.Validator` interface we're going to provide validation behavior for the `Person` class. This is the Validator interface:

- `supports(Class)` - indicates whether or not this validator supports the given object

  `validate(Object, org.springframework.validation.Errors)` - validates the given object and in case of validation errors, put registers those with the given Errors object

Implementing a validator is fairly straightforward, especially when you know of the `ValidationUtils` Spring also provides. Let's review how a validator is created:

```
public class PersonValidator implements Validator {

        public boolean supports(Class clzz) {
                return Person.class.equals(clzz);
        }

        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", "tooold");
                }
        }
}
```

As you can see, the ValidationUtils is used to reject the name property. Have a look at the JavaDoc for ValidationUtils to see what functionality it provides besides the example we gave just now.

5.5. The Errors interface

Validation errors are reported to the Errors object passed to the validator. In case of Spring Web MVC you can use `spring:bind` tags to inspect the error messages, but of course you can also inspect the errors object yourself. The methods it offers are pretty straightforward. More information can be found in the JavaDoc.

5.6. 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, using a `MessageSource`, we're going to output the error messages 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` method 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 register 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", "tooold")`, apart from the `tooold` code, Spring will also register `tooold.age` and `tooold.age.int` (so the first will include the field name and the second will include the type of the field).

More information on the MessageCodesResolver and the default strategy can be found online with the JavaDocs for MessageCodesResolver [http://www.springframework.org/docs/api/org/springframework/validation/MessageCodesResolver.html] and DefaultMessageCodesResolver [http://www.springframework.org/docs/api/org/springframework/validation/DefaultMessageCodesResolver.html] respectively.

Chapter 6. Spring AOP: Aspect Oriented Programming with Spring

6.1. Concepts

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

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

AOP is used in Spring:

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

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

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 for which the implementation might otherwise cut across multiple objects. Transaction management is a good example of a crosscutting concern in J2EE applications. Aspects are implemented using Spring as Advisors or interceptors.
- *Joinpoint*: Point during the execution of a program, such as a method invocation or a particular exception being thrown. In Spring AOP, a joinpoint is always method invocation. Spring does not use the term joinpoint prominently; joinpoint information is accessible through methods on the `MethodInvocation` argument passed to interceptors, and is evaluated by implementations of the `org.springframework.aop.Pointcut` interface.
- *Advice*: Action taken by the AOP framework at a particular joinpoint. Different types of advice include "around," "before" and "throws" advice. Advice types are discussed below. Many AOP frameworks, including Spring, model an advice as an *interceptor*, maintaining a chain of interceptors "around" the joinpoint.

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

Different advice types include:

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

Around advice is the most general kind of advice. Most interception-based AOP frameworks, such as Nanning Aspects, provide only around advice.

As Spring, like AspectJ, provides a full range of advice types, we recommend that you use the least powerful advice type that can implement the required 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 don't need to invoke the `proceed()` method on the MethodInvocation used for around advice, and hence can't fail to invoke it.

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

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 currently supports interception of method invocations. Field interception is not implemented, although support for field interception could be added without breaking the core Spring AOP APIs.
Field interception arguably violates OO encapsulation. We don't believe it is wise in application development. If you require field interception, consider using AspectJ.

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

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

Spring implements the *AOP Alliance* interception interfaces (http://www.sourceforge.net/projects/aopalliance). Around advice must implement the AOP Alliance `org.aopalliance.intercept.MethodInterceptor` interface. Implementations of this interface can run in Spring or any other AOP Alliance compliant implementation. Currently JAC implements the AOP Alliance interfaces, and Nanning and Dynaop are likely to in early 2004.

Spring's approach to AOP differs from that of most other AOP frameworks. The aim is not to provide the most complete AOP implementation (although Spring AOP is quite capable); it is rather to provide a close integration between AOP implementation and Spring IoC to help solve common problems in enterprise applications.

Thus, for example, Spring's AOP functionality is normally used in conjunction with a Spring IoC container. AOP advice is specified using normal bean definition syntax (although this allows powerful "autoproxying" capabilities); advice and pointcuts are themselves managed by Spring IoC: a crucial difference from other AOP implementations. There are some things you can't do easily or efficiently with Spring AOP, such as advise very fine-grained objects. AspectJ is probably the best choice in such cases. However, our experience is that Spring AOP provides an excellent solution to most problems in J2EE applications that are amenable to AOP.

Spring AOP will never strive to compete with AspectJ or AspectWerkz to provide a comprehensive AOP solution. We believe that both proxy-based frameworks like Spring and full-blown frameworks such as AspectJ are valuable, and that they are complementary, rather than in competition. Thus a major priority for Spring 1.1 will be seamlessly integrating 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 will not affect the Spring AOP API or the AOP Alliance API; Spring AOP will remain backward-compatible.

6.1.3. AOP Proxies in Spring

Spring defaults to using J2SE *dynamic proxies* for AOP proxies. This enables any interface or set of interfaces to be proxied.

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

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

6.2. Pointcuts in Spring

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

6.2.1. Concepts

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

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

```
public interface Pointcut {

    ClassFilter getClassFilter();

    MethodMatcher getMethodMatcher();

}
```

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

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

```
public interface ClassFilter {

    boolean matches(Class clazz);
}
```

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

```
public interface MethodMatcher {

    boolean matches(Method m, Class targetClass);

    boolean isRuntime();

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

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

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

6.2.2. 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.

6.2.3. Convenience pointcut implementations

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

6.2.3.1. Static pointcuts

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

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

6.2.3.1.1. Regular expression pointcuts

One obvious way to specific static pointcuts is regular expressions. Several AOP frameworks besides Spring make this possible. `org.springframework.aop.support.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 reference an Advice also (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 serves as both pointcut and advisor, 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.

6.2.3.1.2. Attribute-driven pointcuts

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

6.2.3.2. Dynamic pointcuts

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

The main example is the `control flow` pointcut.

6.2.3.2.1. Control flow pointcuts

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

Note

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

6.2.4. Pointcut superclasses

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

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

6.2.5. Custom pointcuts

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

6.3. Advice types in Spring

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

6.3.1. Advice lifecycles

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

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

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

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

6.3.2. 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.

6.3.2.1. Interception around advice

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

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

```
public interface MethodInterceptor extends Interceptor {

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

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

A simple *MethodInterceptor* implementation looks as follows:

```
public class DebugInterceptor implements MethodInterceptor {

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

```
        System.out.println("Invocation returned");
        return rval;
    }
}
```

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

6.3.2.2. Before advice

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

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

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

```
public interface MethodBeforeAdvice extends BeforeAdvice {

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

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

An example of a before advice in Spring, which counts all 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;
    }
}
```

Before advice can be used with any pointcut.

6.3.2.3. Throws advice

Throws advice is invoked after the return of the joinpoint if the joinpoint threw an exception. Spring offers

typed throws advice. Note that this means that the `org.springframework.aop.ThrowsAdvice` interface does not contain any methods: it is a tag interface identifying that the given object implements one or more typed throws advice methods. These should be of form

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

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

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

```
public class RemoteThrowsAdvice implements ThrowsAdvice {

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

The following advice is invoked if a *ServletException* is thrown. Unlike the above advice, it declares 4 arguments, so that it has access to the invoked method, method arguments and target object:

```
public class ServletThrowsAdviceWithArguments implements ThrowsAdvice {

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

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

```
public static class CombinedThrowsAdvice implements ThrowsAdvice {

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

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

Throws advice can be used with any pointcut.

6.3.2.4. After Returning advice

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

```
public interface AfterReturningAdvice extends Advice {

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

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

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

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {

    private int count;

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

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

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

6.3.2.5. 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 `suppressInterflace(Class intf)` method to suppress interfaces that should not be exposed. However, no matter how many interfaces an `IntroductionInterceptor` is prepared to support, the `IntroductionAdvisor` used will control which interfaces are actually exposed. An introduced interface will conceal any implementation of the same interface by the target.

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

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

```
public class LockMixin extends DelegatingIntroductionInterceptor
    implements Lockable {

    private boolean locked;

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

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

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

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

}
```

Often it isn't necessary to override the `invoke()` method: the `DelegatingIntroductionInterceptor`

implementation--which calls the delegate method if the method is introduced, otherwise proceeds towards the joinpoint--is usually sufficient. In the present case, we need to add a check: no setter method can be invoked if in locked mode.

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

```
public class LockMixinAdvisor extends DefaultIntroductionAdvisor {

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

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

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

6.4. Advisors in Spring

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

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

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

6.5. Using the ProxyFactoryBean to create AOP proxies

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

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

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

6.5.2. JavaBean properties

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

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

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

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

Other properties specific to `ProxyFactory`Bean include:

- `proxyInterfaces`: array of String interface names. If this isn't supplied, a CGLIB proxy for the target class will be used
- `interceptorNames`: String array of Advisor, interceptor or other advice names to apply. Ordering is significant. First come, first serve that is. The first interceptor in the list will be the first to be able to intercept the invocation (of course if it concerns a regular `MethodInterceptor` or `BeforeAdvice`.

  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 advise.

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 below.

- singleton: whether or not the factory should return a single object, no matter how often the `getObject()` method is called. Several `FactoryBean` implementations offer such a method. Default value is true. If you want to use stateful advice--for example, for stateful mixins--use prototype advices along with a singleton value of false.

6.5.3. Proxying interfaces

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

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

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

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

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

6.5.4. Proxying classes

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

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

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

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

weaving in the advice.

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

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

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

6.5.5. 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"/>
```

6.6. Convenient proxy creation

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

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

6.6.1. TransactionProxyFactoryBean

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

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

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

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

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

```
  <!-- Other dependencies omitted -->
</bean>

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

As with the `ProxyFactoryBean`, we might choose to use an inner bean to set the value of `target` property, instead of a reference to a top-level target bean.

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

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

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

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

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

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

6.6.2. EJB proxies

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

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

6.7. 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 to wraps the target of the proxy as an inner bean definition, since the target will never be used on its own anyways.

```
<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. 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 try to pre-instantiate it.

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

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

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

6.9. Manipulating advised objects

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

```
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 introduction).

By default, it's possible to add or remove advisors or interceptors even once a proxy has been created. The only restriction is that it's impossible to add or remove an introduction advisor, as existing proxies from the factory will not show the interface change. (You can obtain a new proxy from the factory to avoid this problem.)

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);
```

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.

6.10. Using the "autoproxy" facility

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

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

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

There are two ways to do this:

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

6.10.1. Autoproxy bean definitions

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

6.10.1.1. 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 interceptor, 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, and with minimal volume of configuration. It is a popular choice for applying declarative transactions to multiple objects.

Bean definitions whose names match, such as "jdkMyBean" and "onlyJdk" in the above example, are plain old bean definitions with the target class. An AOP proxy will be created automatically by the `BeanNameAutoProxyCreator`. The same advice will be applied to all matching beans. Note that if advisors are used (rather than the interceptor in the above example), the pointcuts may apply differently to different beans.

6.10.1.2. 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; default is unordered.

6.10.1.3. 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`.

6.10.2. 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"/>
```

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

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

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

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

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

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

6.11. Using TargetSources

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

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

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

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

6.11.1. 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.

6.11.2. Pooling target sources

Using a pooling target source provides a similar programming model to stateless session EJBs, in which a pool of identical instances is maintained, with method invocations going to free objects in the pool.

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

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

Sample configuration is shown below:

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

<bean id="poolTargetSource" class="org.springframework.aop.target.CommonsPoolTargetSource">
  <property name="targetBeanName" value="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="poolConfig" 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());
```

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.

6.11.3. 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.

6.11.4. 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 target sources:

```
<bean id="threadlocalTargetSource" class="org.springframework.aop.target.ThreadLocalTargetSource">
  <property name="targetBeanName" value="businessObjectTarget"/>
</bean>
```

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 is doing this for you and should always be considered in favor of using ThreadLocals without other proper handling code.

6.12. Defining new Advice types

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

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

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

6.13. Further reading and resources

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

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

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

If you are interested in more advanced capabilities of Spring AOP, take a look at the test suite. The test coverage is over 90%, and this illustrates advanced features not discussed in this document.

Chapter 7. AspectJ Integration

7.1. Overview

Spring's proxy-based AOP framework is well suited for handling many generic middleware and application-specific problems. However, there are times when a more powerful AOP solution is required: for example, if we need to add additional fields to a class, or advise fine-grained objects that aren't created by the Spring IoC container.

We recommend the use of AspectJ in such cases. Accordingly, as of version 1.1, Spring provides a powerful integration with AspectJ.

7.2. Configuring AspectJ aspects using Spring IoC

The most important part of the Spring/AspectJ integration allows Spring to configure AspectJ aspects using Dependency Injection. This brings similar benefits to aspects as to objects. For example:

- There is no need for aspects to use ad hoc configuration mechanisms; they can be configured in the same, consistent, approach used for the entire application.
- Aspects can depend on application objects. For example, a security aspect can depend on a security manager, as we'll see in an example shortly.
- It's possible to obtain a reference to an aspect through the relevant Spring context. This can allow for dynamic reconfiguration of the aspect.

AspectJ aspects can expose JavaBean properties for Setter Injection, and even implement Spring lifecycle interfaces such as `BeanFactoryAware`.
Note that AspectJ aspects cannot use Constructor Injection or Method Injection. This limitation is due to the fact that aspects do not have constructors that can be invoked like constructors of objects.

7.2.1. "Singleton" aspects

In most cases, AspectJ aspects are singletons, with one instance per class loader. This single instance is responsible for advising multiple object instances.

A Spring IoC container cannot instantiate an aspect, as aspects don't have callable constructors. But it can obtain a reference to an aspect using the static `aspectOf()` method that AspectJ defines for all aspects, and it can inject dependencies into that aspect.

7.2.1.1. Example

Consider a security aspect, which depends on a security manager. This aspects applies to all changes in the value of the `balance` instance variable in the `Account` class. (We couldn't do this in the same way using Spring AOP.)

The AspectJ code for the aspect (one of the Spring/AspectJ samples), is shown below. Note that the dependency on the `SecurityManager` interface is expressed in a JavaBean property:

```
public aspect BalanceChangeSecurityAspect {

   private SecurityManager securityManager;

   public void setSecurityManager(SecurityManager securityManager) {
      this.securityManager = securityManager;
   }

   private pointcut balanceChanged() :
      set(int Account.balance);

   before() : balanceChanged() {
      this.securityManager.checkAuthorizedToModify();
   }
}
```

We configure this aspect in the same way as an ordinary class. Note that the way in which we set the property reference is identical. Note that we must use the `factory-method` attribute to specify that we want the aspect "created" using the `aspectOf()` static method. In fact, this is *locating*, rather than, `creating`, the aspect, but the Spring container doesn't care:

```
<bean id="securityAspect"
   class="org.springframework.samples.aspectj.bank.BalanceChangeSecurityAspect"
   factory-method="aspectOf"
>
   <property name="securityManager" ref="securityManager"/>
</bean>
```

We don't need to do anything in Spring configuration to target this aspect. It contains the pointcut information in AspectJ code that controls where it applies. Thus it can apply even to objects not managed by the Spring IoC container.

7.2.1.2. Ordering issues

to be completed

7.2.2. Non-singleton aspects

** Complete material on pertarget etc.

7.2.3. Gotchas

to be completed

- Singleton issue

7.3. Using AspectJ pointcuts to target Spring advice

In a future release of Spring, we plan to provide the ability for AspectJ pointcut expressions to be used in Spring XML or other bean definition files, to target Spring advice. This will allow some of the power of the AspectJ pointcut model to be applied to Spring's proxy-based AOP framework. This will work in pure Java, and will not require the AspectJ compiler. Only the subset of AspectJ pointcuts relating to method invocation will be usable.

This feature replaces our previous plan to create a pointcut expression language for Spring.

7.4. Spring aspects for AspectJ

In a future release of Spring, we will package some Spring services, such as the declarative transaction management service, as AspectJ aspects. This will enable them to be used by AspectJ users without dependence on the Spring AOP framework--potentially, even without dependence on the Spring IoC container.

This feature is probably of more interest to AspectJ users than Spring users.

Chapter 8. Transaction management

8.1. The Spring transaction abstraction

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

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

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

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

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

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

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

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

8.2. Transaction strategies

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

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

```
public interface PlatformTransactionManager {

    TransactionStatus getTransaction(TransactionDefinition definition)
        throws TransactionException;

    void commit(TransactionStatus status) throws TransactionException;

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

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

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

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

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

The `TransactionDefinition` interface specifies:

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

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

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

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

```
public interface TransactionStatus {

    boolean isNewTransaction();

    void setRollbackOnly();

    boolean isRollbackOnly();
}
```

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

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

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

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

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

The PlatformTransactionManager definition will look like this:

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

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

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

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

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

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

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

The "txManager" bean in this case is of class HibernateTransactionManager. In the same way as the DataSourceTransactionManager needs a reference to the DataSource, the HibernateTransactionManager needs a reference to the 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">
    <props>
      <prop key="hibernate.dialect">${hibernate.dialect}</prop>
    </props>
  </property>
</bean>

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

With Hibernate and JTA transactions we could 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 won't need to change at all. We can change how transactions are managed merely by changing configuration, even if that change means moving from local to global transactions or vice versa.

8.3. 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 to trigger (optionally) transaction synchronization via the relevant `PlatformTransactionManager`.

8.3.1. High-level approach

The preferred approach is to use Spring's highest level persistence integration APIs. These do not replace the native APIs, but do 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 followed for all persistence APIs, with classes such as `JdbcTemplate`, `HibernateTemplate`, `JdoTemplate`, etc.. These integration classes are detailed in subsequent chapters of this manual.

8.3.2. 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 preferred for application code to deal directly with the resource types of the native persistence APIs, these classes ensure that proper Spring-managed instances are obtained, transactions are (optionally) synchronized to, and exceptions which happen in the process are properly mapped to a consistent API.

For example, for 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 available for subsequent reuse in that same transaction. As mentioned, this has the added advantage that any SQLException will be wrapped in a Spring CannotGetJdbcConnectionException--one of Spring's hierarchy of unchecked DataAccessExceptions. This gives you more information than can easily be obtained from the SQLException, and ensures portability across databases: even across different persistence technologies.

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.

All these lower level resource access classes are detailed in subsequent chapters of this manual.

8.3.3. TransactionAwareDataSourceProxy

At the very lowest level exists the TransactionAwareDataSourceProxy class. This 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.

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 preferable to write your own new code using the higher level abstractions mentioned above.

See the TransactionAwareDataSourceProxy Javadocs for more details.

8.4. Programmatic transaction management

Spring provides two means of programmatic transaction management:

- Using the TransactionTemplate
- Using a PlatformTransactionManager implementation directly

We generally recommend the first approach.

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

8.4.1. Using the `TransactionTemplate`

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

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

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

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

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

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

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

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

8.4.2. 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();
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);
```

8.5. Declarative transaction management

Spring also offers declarative transaction management. This is enabled by Spring AOP, although, as the transactional aspects code comes with Spring and may be used in a boilerplate fashion, AOP concepts do not generally have to be understood to make effective use of this code..
Most Spring users 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.

It may be helpful to begin by considering EJB CMT and explaining the similarities and differences with Spring declarative transaction management. The basic approach is similar: It's possible to specify transaction behavior (or lack of it) down to individual methods. It's possible to make a `setRollbackOnly()` call within a transaction context if necessary. The differences are:

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

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

While the EJB default 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* (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's often useful to customize this.

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

The usual way of setting up transactional proxying in Spring is via the the use of

`TransactionProxyFactoryBean` to create the transactional proxy. This factory bean is simply a specialized version of Spring's generic `ProxyFactoryBean`, that, in addition to creating a proxy to wrap a target object, will also always automatically create and attach a `TransactionInterceptor` to that proxy, reducing boilerplate code. (Note that as with `ProxyFactoryBean`, you may still specify other interceptors or AOP advice to apply via the proxy).

When using `TransactionProxyFactoryBean`, you need to first of all specify the target object to wrap in the transactional proxy, via the `target` attribute.. The target object is normally a POJO bean definition. You must also specify a reference to the relevant `PlatformTransactionManager`. Finally, you must specify the **transaction attributes**. Transaction attributes contain the definition of what transaction semantics we wish to use (as discussed above), as well as where they apply. Now let's consider the following sample:

```
<!-- this example is in verbose form, see note later about concise for multiple proxies! -->
<!-- the target bean to wrap transactionally -->
<bean id="petStoreTarget">
  ...
</bean>

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

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

The transaction interceptor will ultimately use an object implementating Spring's `TransactionAttributeSource` interface to get at the transaction attributes (in the form of `TransactionAttribute` objects) defining the transaction semantics to be applied to specific methods of specific classes. The most basic way to specify this `TransactionAttributeSource` instance when creating the proxy is for you to create a bean implementing the `TransactionAttributeSource` interface (Spring has several implementations), and then directly set the the `transactionAttributeSource` property of the proxy factory bean to refer to it (or wrap it as an inner bean. Alternately, you may set a text string for this property, and rely on the fact that the pre-registered (by Spring) `TransactionAttributeSourceEditor` will automatically convert that text string to a `MethodMapTransactionAttributeSource` instance.

However, as shown in this example, most users will instead prefer to define the transaction attributes by setting the `transactionAttributes` property. This property has a type of `Java.util.Properties`, which will then internally be converted to a `NameMatchTransactionAttributeSource` object.

As can be seen in the above definition, a `NameMatchTransactionAttributeSource` object holds a list of name/value pairs. The key of each pair is a method or methods (a * wildcard ending is optional) to apply transactional semantics to. Note that the method name is *not* qualified with a package name, but rather is considered relative to the class of the target object being wrapped. The value portion of the name/value pair is the `TransactionAttribute` itself that needs to be applied. When specifying it as the `Properties` value as in

this example, it's in String format as defined by `TransactionAttributeEditor`. This format is:

```
PROPAGATION_NAME,ISOLATION_NAME,readOnly,timeout_NNNN,+Exception1,-Exception2
```

Note that the only mandatory portion of the string is the propagation setting. The default transactions semantics which apply are as follows:

- Exception Handling: RuntimeExceptions roll-back, normal (checked) Exceptions don't
- Transactions are read/write
- Isolation Level: TransactionDefinition.ISOLATION_DEFAULT
- Timeout: TransactionDefinition.TIMEOUT_DEFAULT

See the JavaDocs for `org.springframework.transaction.TransactionDefinition` class for the format allowed for the propagation setting and isolation level setting. The String format is the same as the Integer constant names for the same values.

In this example, note that the value for the insert* mapping contains a rollback rule. Adding `-MyCheckedException` here specifies that if the method throws `MyCheckedException` or any subclasses, the transaction will automatically be rolled back. Multiple rollback rules can be specified here, comma-separated. A - prefix forces rollback; a + prefix specifies commit. (This allows commit even on unchecked exceptions, if you really know what you're doing!)

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

Note: Using TransactionProxyFactoryBean definitions in the form above can seem overly verbose when many almost identical transaction proxies need to be created. You will almost always want to take advantage of parent and child bean definitions, along with inner bean definitions, to significantly reduce the verbosity of your transaction proxy definitions, as described in Section 6.7, "Concise proxy definitions".

8.5.1. Source Annotations for Transaction Demarcation

XML-based transaction attribute sources definitions are convenient, and work in any environment, but if you are willing to commit to a dependency on Java 5+ (JDK 1.5+), you will almost certainly want to consider using Spring's support for transaction Annotations in JDK standard format, as the attribute source instead.

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 typically, code that is deployed as transactional is always deployed that way.

8.5.1.1. The `Transactional` Annotation

The `org.springframework.transaction.annotation.Transactional Annotation` is used to indicate that an

interface, interface method, class, or class method should have transaction semantics.

```
@Transactional
public interface OrderService {

void createOrder(Order order);
List queryByCriteria(Order criteria);
```

Used in bare form, this Annotation specifies that an interface, class, or method must be transactional. Default transaction semantics are read/write, PROPAGATION_REQUIRED, ISOLATION_DEFAULT, TIMEOUT_DEFAULT, with rollback on a `RuntimeException`, but not `Exception`.

Optional properties of the annotation modify transaction settings.

Table 8.1. Properties of the `Transactional` Annotation

Property	Type	Description
`propagation`	enum: Propagation	optional propagation setting (defaults to PROPAGATION_REQUIRED)
`isolation`	enum: Isolation	optional isolation level (defaults to ISOLATION_DEFAULT)
`readOnly`	boolean	read/write vs. read-only transaction (defaults to false, or read/write)
`rollbackFor`	array of Class objects, must be derived from Throwable	optional array of exception classes which should cause rollback. By default, checked exceptions do not roll back, unchecked (RuntimeException derived) roll back
`rollbackForClassname`	array of String class names. Classes must be derived from Throwable	optional array of names of exception classes which should cause rollback
`noRollbackFor`	array of Class objects, must be derived from Throwable	optional array of exception classes which should not cause rollback.
`noRollbackForClassname`	array of String class names, must be derived from Throwable	optional array of names of exception classes which should not rollback

The annotation may be placed before an interface definition, a method on an interface, a class definition, or a method on a class. It may exist on both an element of an interface, and a class which implements that interface. The most derived location takes precedence when evaluating the transaction semantics of a method.

8.5.1.1.1. `Transactional` annotation examples

Annotating a class definition:

```
public class OrderServiceImpl implements OrderService {
```

```
  @Transactional
  void createOrder(Order order);
  public List queryByCriteria(Order criteria);

}
```

In the following example, the interface is annotated for read-only transactions, which will thus be the setting used for methods by default. The Annotation on the createOrder method overrides this, setting the transaction to read/write, and specifying that transactions should also (in addition to the defualt rollback rule for `RuntimeException`) rollback when the `DuplicateOrderIdException` (presumably a non-checked Exception) is thrown.

```
@Transactional(readOnly=true)
interface TestService {

  @Transactional(readOnly=false,
             rollbackFor=DuplicateOrderIdException.class)
  void createOrder(Order order) throws DuplicateOrderIdException ;

  List queryByCriteria(Order criteria);
}
```

Note that a class definition which implements this interface may still override these settings on its own class or method elements.

8.5.1.1.2. Telling Spring to apply the `Transactional` annotation

By itself, adding instances of this annotation to interface or class elements will not result in transactional wrapping of the implementation clases. Spring must still be told somehow to create transactional proxies around classes with these annotations.

The key is to take advantage of the `org.springframework.transaction.annotation.AnnotationTransactionAttributeSource` class, which reads Annotations format transaction attributes from class files. Taking the previous example which uses TransactionProxyFactoryBean, the `TransactionAttributes` property which specified transaction attributes in text form is replaced by the direct usage of the `TransactionAttributeSource` property, specifying an `AnnotationTransactionAttributeSource`.

```
<bean id="petStore" class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager" ref="txManager"/>
  <property name="target" ref="petStoreTarget"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
  </property>
</bean>
```

Since the TransactionAttributeSource property does not need to change at all for each proxy instance, when using parent and child bean definitions to avoid code duplication, the property may just be set on the base, parent definition and forgotten, there is never a need to override it in the child since the attribute source will read the right settings from each class file.

8.5.1.1.3. Using AOP to ensure the `Transactional` annotation is applied

The previous example is still more work than would be ideal. There is in principle no need for XML for each proxy (to point to the target bean) when the annotations in the class files themselves can be used as an indication that a proxy needs to be created for the annotated classes.

A more AOP focused approach allows a small amount of boilerplate XML (used once only, not for each target

bean) to automatically ensure that proxies are created for all classes with Transactional annotations in them. Spring AOP was fully detailed in a previous chapter, which you should consult for general AOP documentation, but the key is the use of `DefaultAdvisorAutoProxyCreator`, a `BeanPostProcessor`. Because it is a bean post processor, it gets a chance to look at every bean that is created as it is created. If the bean contains the `Transactional` annotation, a transactional proxy is automatically created to wrap it.

```
<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="txManager"/>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.annotation.AnnotationTransactionAttributeSource"/>
  </property>
</bean>
```

A number of classes are involved here:

- `TransactionInterceptor`: the AOP Advice, actually intercepts method call and wraps it with a transaction
- `TransactionAttributeSourceAdvisor`: AOP Advisor (holds the TransactionInterceptor, which is the advice, and a pointcut (where to apply the advice), in the form of a TransactionAttributeSource)
- `AnnotationTransactionAttributeSource`: TransactionAttributeSource implementation which provides transaction attributes read from class files
- `DefaultAdvisorAutoProxyCreator`: looks for Advisors in the context, and automatically creates proxy objects which are the transactional wrappers

8.5.2. BeanNameAutoProxyCreator, another declarative approach

`TransactionProxyFactoryBean` is very useful, and gives you full control when wrapping objects with a transactional proxy. Used with parent/child bean definitions and inner beans holding the target, and when Java 5 Annotations are not available as an option, it is generally the best choice for transactional wrapping. In the case that you need to wrap a number of beans in a completely identical fashion (for example, a boilerplate, 'make all methods transactional', using a `BeanFactoryPostProcessor` called `BeanNameAutoProxyCreator` can offer an alternative approach which can end up being even less verbose for this simplified use case.

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

Let us consider a sample configuration:

```
<beans>

        <!-- Transaction Interceptor set up to do PROPAGATION_REQUIRED on all methods -->
  <bean id="matchAllWithPropReq"
      class="org.springframework.transaction.interceptor.MatchAlwaysTransactionAttributeSource">
    <property name="transactionAttribute" value="PROPAGATION_REQUIRED"/>
  </bean>
```

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

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

</beans>
```

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

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

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

There is one thing to keep in mind, with regards to bean naming, when switching back and forth between the use of `TransactionProxyFactoryBean`, and `BeanNameAutoProxyCreator`. For the former, if the target bean is not defined as an inner bean, you normally give the target bean you want to wrap an id similar in form to *myServiceTarget*, and then give the proxy object an id of *myService*; then all users of the wrapped object simply refer to the proxy, i.e. *myService*. (These are just sample naming conventions, the point is that the target object has a different name than the proxy, and both are available from the ApplicationContext). However, when using `BeanNameAutoProxyCreator`, you name the target object something like *myService*. Then, when `BeanNameAutoProxyCreator` postprocesses the target object and create the proxy, it causes the proxy to be inserted into the Application context under the name of the original bean. From that point on, only the proxy (the wrapped object) is available from the ApplicationContext. When using TransactionProxyFactoryBean with the target specified as an inner bean, this naming issue is not a concern, since the inner bean is not normally given a name.

8.5.3. AOP and Transactions

As you've seen by reading this chapter, you don't really need to be an AOP expert--or indeed, to know much at all about AOP--to use Spring's declarative transaction management effectively. However, if you do want to become a "power user" of Spring AOP, you will find it easy to combine declarative transaction management with powerful AOP capabilities.

8.6. Choosing between programmatic and declarative transaction management

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

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

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

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

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

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

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

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

8.8. AppServer-specific integration

Spring's transaction abstraction is generally AppServer 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 AppServer. Having access to the `TransactionManager` instance does allow enhanced transaction semantics. Please see the `JtaTransactionManager` Javadocs for more details.

8.8.1. 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.

Please see the Javadocs for full details.

8.8.2. 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.

8.9. Common problems

8.9.1. Use of the wrong transaction manager for a specific DataSource

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

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

If you are using global transactions, you *must* use the Spring `org.springframework.transaction.jta.JtaTransactionManager` for all your for all your transactional operations. Otherwise Spring will attempt to perform local transactions on resources such as container DataSources. Such local transactions don't make sense, and a good application server will treat them as errors.

8.9.2. Spurious AppServer warnings about the transaction or DataSource no longer being active

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

`TransactionManager` object for that environment, it is 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 as described in Section 12.2.10, “Spurious AppServer warnings about the transaction or DataSource no longer being active”.

Chapter 9. Source Level Metadata Support

9.1. Source-level metadata

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

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

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

We could add metadata to a method as follows:

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

Both 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 exclusively used 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 primarily view metadata 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 already supports specific Java 5 Annotations for transactional demarcation, and for JMX. But we need a solution for Java 1.4 and even 1.3 too.
- **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 appears to be the most promising. All these require a special pre- or post-compilation step.

9.2. 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:

- 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. 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 aims to provide the subset of metadata required to provide effective declarative enterprise services a la EJB or .NET, on Java 1.3+. As of Spring 1.2, analogous JSR-175 annotations are supported on JDK 1.5, as 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 and JBoss 4 (as of DR2), which offer only `String` attributes. There is a significant advantage in supporting `Object` attributes. It enables attributes to participate in class hierarchies and enables attributes to react intelligently to their configuration parameters.

In most attribute providers, attribute classes will be configured via constructor arguments or JavaBean properties. Commons Attributes supports both.

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

9.3. Integration with Jakarta Commons Attributes

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

Commons Attributes 2.1 (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 an FQN, as shown above. If the relevant attribute class has already been imported, the FQN isn't required. It's also possible to specify "attribute packages" in attribute compiler configuration.
- *Any necessary parameterization,* via constructor arguments or JavaBean properties

Bean properties look as follows:

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

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

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

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

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

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

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

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

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

```
<target name="compileAttributes">

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

</target>
```

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

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

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

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

  <classpath refid="master-classpath"/>

</attribute-indexer>
```

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

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

9.4. Metadata and Spring AOP autoproxying

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

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

9.4.1. 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 those 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 do know about attributes.

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

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

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

9.4.2. Declarative transaction management

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

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

9.4.3. Pooling

Again, as with .NET, you can 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)
 * @author Rod Johnson
 */
public class MyClass {
```

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

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

9.4.4. Custom metadata

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

We can define custom attributes, to provide any kind of declarative 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.)

9.5. Using attributes to minimize MVC web tier configuration

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

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

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

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

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

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

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

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

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

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

    private Cruncher cruncher;

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

    protected ModelAndView handleRequestInternal(
            HttpServletRequest 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
* @author Rod Johnson
*
* @@org.springframework.web.servlet.handler.metadata.PathMap("/foo.cgi")
* @@org.springframework.web.servlet.handler.metadata.PathMap("/baz.cgi")
*/
public class FooController extends AbstractController {

    private Cruncher cruncher;

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

    protected ModelAndView handleRequestInternal(
            HttpServletRequest 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 more complex build process. We need an attribute compilation step and an attribute

indexing step. However, once in place, this should not be an issue.

- Currently Commons Attributes only, although support for other attribute providers may be added in future.
- Only "autowiring by type" dependency injection is supported for such controllers. However, this still leaves them far in advance of Struts Actions (with no IoC support from the framework) and, arguably, WebWork Actions (with only rudimentary IoC support) where IoC is concerned.
- Reliance on automagical IoC resolution may be confusing.

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

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

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

9.6. Other uses of metadata attributes

Other uses of metadata attributes appear to be growing in popularity. As of Spring 1.2, metadata attributes for JMX exposure are supported, through both Commons Attributes (on JDK 1.3+) and JSR-175 annotations (on JDK 1.5).

9.7. Adding support for additional metadata APIs

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

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

All framework services that use metadata, such as AOP metadata-driven autoproxying, will then automatically be able to use your new metadata provider.

Chapter 10. DAO support

10.1. Introduction

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

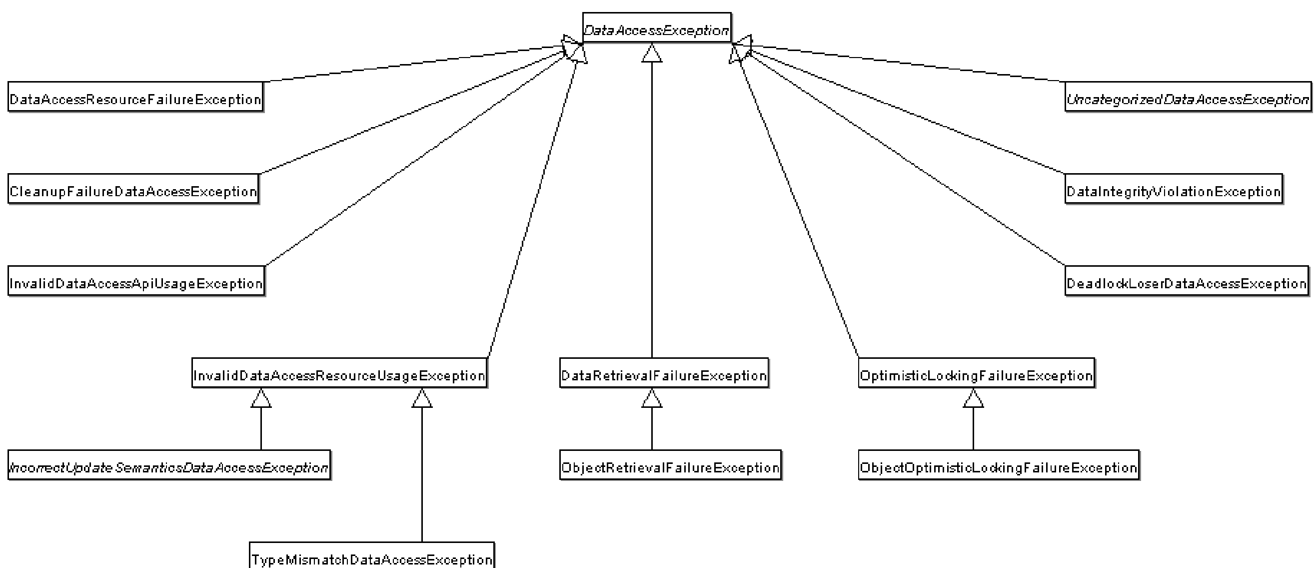
10.2. Consistent Exception Hierarchy

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

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

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

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



10.3. Consistent Abstract Classes for DAO Support

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

Dao Support classes:

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

Chapter 11. Data Access using JDBC

11.1. Introduction

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

The `org.springframework.jdbc.core` package 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

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

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

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

11.2.2. DataSource

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

11.2.3. SQLExceptionTranslator

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

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

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

The `SQLErrorCodeSQLExceptionTranslator` applies the following matching rules:

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

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

`SQLErrorCodeSQLExceptionTranslator` can be extended the following way:

```
public class 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`.

11.2.4. Executing Statements

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

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

public class ExecuteAStatement {

    private JdbcTemplate jt;
    private DataSource dataSource;

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

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

11.2.5. Running Queries

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

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

public class RunAQuery {

    private JdbcTemplate jt;
    private DataSource dataSource;

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

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

    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 one is `queryForList` which returns a `List` where each entry is a `Map` with each entry in the map representing the column value for that row. If we add a method to the above example to retrieve a list of all the rows, it would look like this:

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

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

11.2.6. Updating the database

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

```
import javax.sql.DataSource;

import org.springframework.jdbc.core.JdbcTemplate;

public class ExecuteAnUpdate {

    private JdbcTemplate jt;
    private DataSource dataSource;

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

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

11.3. Controlling how we connect to the database

11.3.1. DataSourceUtils

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

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

11.3.2. SmartDataSource

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

11.3.3. AbstractDataSource

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

11.3.4. SingleConnectionDataSource

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

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

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

11.3.5. DriverManagerDataSource

Implementation of `SmartDataSource` 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 respective ApplicationContext, or in conjunction with a simple JNDI environment. Pool-assuming `Connection.close()` calls will simply close the connection, so any DataSource-aware persistence code should work. 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

This 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.

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

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

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

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

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

11.4. Modeling JDBC operations as Java objects

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

11.4.1. SqlQuery

Reusable thread safe object to represent an SQL query. Subclasses must implement the newResultReader()

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

11.4.2. MappingSqlQuery

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

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

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

```
private class CustomerMappingQuery extends MappingSqlQuery {

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

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

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

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

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

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

11.4.3. SqlUpdate

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

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

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

```
import java.sql.Types;

import javax.sql.DataSource;

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

public class UpdateCreditRating extends SqlUpdate {

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

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

11.4.4. StoredProcedure

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

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

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

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

import javax.sql.DataSource;

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

```
public class 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 res = sproc.execute();
        printMap(res);
    }

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

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

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

    }

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

11.4.5. SqlFunction

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

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

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

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

Chapter 12. Data Access using O/R Mappers

12.1. Introduction

Spring provides integration with *Hibernate, JDO, Oracle TopLink, Apache OJB* and *iBATIS SQL Maps*: 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 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's 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 an ApplicationContext does provide additional benefits in terms of ease of configuration and deployment; as such, most examples in this section show configuration inside an ApplicationContext.

Some of the the benefits of using Spring to create your O/R mapping DAOs include:

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

- *Common data access exceptions.*Spring can wrap exceptions from you 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 DataSources, 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 Hibernate3 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 O/R mapping operations.

- *To avoid vendor lock-in, and allow mix-and-match implementation strategies.* While Hibernate is powerful, flexible, open source and free, it still uses a proprietary API. Furthermore one could argue that iBATIS is a bit lightweight, although it's excellent for use in application that don't require complex O/R mapping strategies. Given the choice, it's usually desirable to implement major application functionality using standard or abstracted APIs, in case you need to switch to another implementation for reasons of functionality, performance, or any other concerns. For example, Spring's abstraction of Hibernate transactions and exceptions, along with its IoC approach which allows you to easily swap in mapper/DAO objects implementing data access functionality, makes it easy to isolate all Hibernate-specific code in one area of your application, without sacrificing any of the power of Hibernate. Higher level service code dealing with the DAOs has no need to know anything about their implementation. This approach has the additional benefit of making it easy to intentionally implement data access with a mix-and-match approach (i.e. some data access performed using Hibernate, and some using JDBC, others using iBATIS) in a non-intrusive fashion, potentially providing great benefits in terms of continuing to use legacy code or leveraging the strength of each technology.

The PetClinic sample in the Spring distribution offers alternative DAO implementations and application context configurations for JDBC, Hibernate, Oracle TopLink, and Apache OJB. PetClinic can therefore serve as working sample app that illustrates the use of Hibernate, TopLink and OJB 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 is 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.solarmetric.com).

12.2. Hibernate

We will start with a coverage of Hibernate (http://www.hibernate.org) 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 O/R mapping tools. The following sections in this chapter will then cover the other O/R mappers, showing briefer examples there.

The following discussion focuses on "classic" Hibernate: that is, Hibernate 2.1, which has been supported in Spring since its inception. All of this can be applied to Hibernate 3.0 as-is, using the analogous Hibernate 3 support package introduced in Spring 1.2 final: `org.springframework.orm.hibernate3`, mirroring `org.springframework.orm.hibernate` with analogous support classes for Hibernate 3. Furthermore, all references to the `net.sf.hibernate` package need to be replaced with `org.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: Inversion of control via templating, i.e. infrastructure classes with callback interfaces, or applying AOP interceptors. The infrastructure cares for proper resource handling, and for appropriate conversion of specific API exceptions to an unchecked infrastructure exception hierarchy. Spring introduces a DAO exception hierarchy, applicable to any data access strategy. For direct JDBC, the `JdbcTemplate` class mentioned in a previous section cares for connection handling, and for proper conversion of `SQLException` to the `DataAccessException` hierarchy, including translation of database-specific SQL error codes to meaningful exception classes. It supports both JTA and JDBC transactions, via respective Spring transaction managers.

Spring also offers Hibernate and JDO support, consisting of a `HibernateTemplate` / `JdoTemplate` analogous to `JdbcTemplate`, a `HibernateInterceptor` / `JdoInterceptor`, and a Hibernate / JDO transaction manager. The major goal is to allow for clear application layering, with any data access and transaction technology, and for loose coupling of application objects. No more business 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), etc.

12.2.2. SessionFactory setup in a Spring application context

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

```
<beans>

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

   ...
</beans>
```

Note that switching from a local Jakarta Commons DBCP `BasicDataSource` to a JNDI-located `DataSource`

(usually managed by the J2EE 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's typically not necessary outside an EJB context. See the "container resources versus local resources" section for a discussion.

12.2.3. Inversion of Control: HibernateTemplate and HibernateCallback

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 application context - via a simple `setSessionFactory` bean property setter. The following snippets show a DAO definition in a Spring application context, referencing the above defined `SessionFactory`, and an example for a DAO method implementation.

```
<beans>
  ...

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

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

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

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        HibernateTemplate ht = new HibernateTemplate(this.sessionFactory);
        return (Collection) ht.execute(new HibernateCallback() {
            public Object doInHibernate(Session session) throws HibernateException {
                Query query = session.createQuery(
                     "from test.Product product where product.category=?");
                query.setString(0, category);
                return query.list();
            }
        });
    }
}
```

A callback implementation can effectively be used for any Hibernate data access. `HibernateTemplate` will ensure that `Session`s 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 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 complying to Spring's generic `DataAccessException` hierarchy. Spring's `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" into `getSession`'s the "allowCreate" flag, to enforce running within a transaction (which avoids the need to close the returned Session, as it its lifecycle is managed by the transaction).

```
public class ProductDaoImpl extends HibernateDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category)
            throws DataAccessException, MyException {

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

The major advantage of such direct Hibernate access code is that it allows any checked application exception to be thrown within the data access code, while `HibernateTemplate` is restricted to unchecked exceptions within the callback. Note that one can often defer the corresponding checks and the throwing of application exceptions to after the callback, which still allows working with `HibernateTemplate`. In general, `HibernateTemplate`'s convenience methods are simpler and more convenient for many scenarios.

12.2.5. Implementing DAOs based on plain Hibernate3 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 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 Hibernate access style is very similar to what you will find in the Hibernate 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!)

Our DAO above follows the Dependency Injection pattern: It still fits nicely into a Spring application context, just like it would if coded against Spring's `HibernateTemplate`. Concretely, it uses Setter Injection; if desired, it could use Constructor Injection instead. 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 might 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 tying 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.

A further disadvantage of that DAO style is that Hibernate's `getCurrentSession()` feature just works within JTA transactions. It does not work with any other transaction strategy out-of-the-box, in particular not with local Hibernate transactions.

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 plain JTA).

In summary: DAOs can be implemented based on plain Hibernate3 API, while still being able to participate in Spring-managed transactions. This might in particular appeal to people already familar with Hibernate, feeling more natural to them. However, such DAOs will throw plain `HibernateException`; conversion to Spring's `DataAccessException` would have to happen explicitly (if desired).

12.2.6. Programmatic transaction demarcation

On top of such lower-level data access services, transactions can be demarcated in a higher level of the application, spanning any number of operations. There are no restrictions on the implementation of the surrounding business 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.hibernate.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 PlatformTransactionManager transactionManager;
    private ProductDao productDao;

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

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

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

12.2.7. Declarative transaction demarcation

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

```
<beans>
    ...

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

  <bean id="myTxInterceptor"
      class="org.springframework.transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="transactionAttributeSource">
      <value>
        product.ProductService.increasePrice*=PROPAGATION_REQUIRED
        product.ProductService.someOtherBusinessMethod=PROPAGATION_MANDATORY
      </value>
    </property>
```

```
  </bean>

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

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

</beans>
```

```
public class ProductServiceImpl implements ProductService {

    private ProductDao productDao;

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

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

    ...
}
```

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. A convenient alternative way of setting up declarative transactions is `TransactionProxyFactoryBean`, particularly if there are no other AOP interceptors involved. `TransactionProxyFactoryBean` combines the proxy definition itself with transaction configuration for a particular target bean. This reduces the configuration effort to one target bean plus one proxy bean. Furthermore, you do not need to specify which interfaces or classes the transactional methods are defined in.

```
<beans>
  ...

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

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

  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target" ref="myProductServiceTarget"/>
    <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>
```

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, i.e. when facing distributed transaction requirements for certain deployments of your application, is just a matter of configuration. Simply replace the Hibernate transaction manager with Spring's JTA transaction implementation. Both transaction demarcation and data access code will work without changes, as they just use the generic transaction management APIs.

For distributed transactions across multiple Hibernate session factories, simply combine `JtaTransactionManager` as a transaction strategy with multiple `LocalSessionFactoryBean` definitions. Each of your DAOs then gets one specific SessionFactory reference passed into its respective bean property. If all underlying JDBC data sources are transactional container ones, a business 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.hibernate.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource1"/>
    <property name="mappingResources">
      <list>
        <value>product.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.MySQLDialect</prop>
      </props>
    </property>
  </bean>

  <bean id="mySessionFactory2" class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
    <property name="dataSource" ref="myDataSource2"/>
    <property name="mappingResources">
      <list>
        <value>inventory.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">net.sf.hibernate.dialect.OracleDialect</prop>
      </props>
    </property>
  </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>

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

  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target" ref="myProductServiceTarget"/>
    <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 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 LocalSessionFactoryBean's "dataSource" property). Alternatively, the DataSource that the transactions are supposed to be exposed for can also be specified explicitly, through HibernateTransactionManager's "dataSource" property.

Note, for an alternative approach to using `TransactionProxyFactoryBean` to declaratively demarcate transactions, please see Section 8.5.2, “BeanNameAutoProxyCreator, another declarative approach”.

12.2.9. Container resources versus local resources

Spring's resource management allows for simple switching between a JNDI SessionFactory and a local one, same for a JNDI DataSource, without having to change a single line of application code. Whether to keep the resource definitions in the container or locally within the application, is mainly a matter of the transaction strategy being used. Compared to a Spring-defined local SessionFactory, a manually registered JNDI SessionFactory does not provide any benefits. 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 DataSources. 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 e.g. WebLogic Express does not provide JCA. A Spring app with local resources and transactions spanning one single database will work in any J2EE web container (without JTA, JCA, or EJB) - like Tomcat, Resin, or even plain Jetty. Additionally, such a middle tier can be reused in desktop applications or test suites easily.

All things considered: If you do not use EJB, stick with local SessionFactory setup and Spring's `HibernateTransactionManager` or `JtaTransactionManager`. You will get all benefits including proper transactional JVM-level caching and distributed transactions, without any container deployment hassle. JNDI registration of a Hibernate SessionFactory via the JCA connector only adds value for use within EJBs.

12.2.10. Spurious AppServer warnings about the transaction or DataSource no longer being active

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 `TransactionManager` object for that environment, it is 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 `TransactionManager` 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 `TransactionManager` 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 `TransactionManager` 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, bu the full sequence of events with and without Hibernate being aware of the JTA `TransactionManager` will now be described.

When Hibernate is not configured with any awareness of the JTA `TransactionManager`, 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 `TransactionManager`, 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 senario. 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, i.e. through a bean property setter or through a constructor argument. Such a DAO could be coded against plain JDO API, working with the given `PersistenceManagerFactory`, but will usually rather be used with Spring's `JdoTemplate`:

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private PersistenceManagerFactory persistenceManagerFactory;

    public void setPersistenceManagerFactory(PersistenceManagerFactory pmf) {
        this.persistenceManagerFactory = pmf;
    }

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        JdoTemplate jdoTemplate = new JdoTemplate(this.persistenceManagerFactory);
        return (Collection) 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 `PersistenceManager`s 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, explictly 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 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 application context, 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 transaction `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 familar 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, you could define a `TransactionProxyFactoryBean` for a ProductService, which in turn delegates to the JDO-based ProductDao. Each specified method would then automatically get executed within a transaction, with all affected DAO operations automatically participating in it.

```
<beans>
  ...

  <bean id="myTxManager" class="org.springframework.orm.jdo.JdoTransactionManager">
    <property name="persistenceManagerFactory" ref="myPmf"/>
  </bean>

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

  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target" ref="myProductServiceTarget"/>
    <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>
```

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 `JdoTransactionManager` 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`s 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 `Session`s 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 ProductDaoImpl implements ProductDao {

    private SessionFactory sessionFactory;

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

    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        TopLinkTemplate tlTemplate = new TopLinkTemplate(this.sessionFactory);
        return (Collection) 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 `Session`s 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, explictly 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 familar 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, you could define a `TransactionProxyFactoryBean` for a ProductService, which in turn delegates to the TopLink-based ProductDao. Each specified method would then automatically get executed within a transaction, with all affected DAO operations automatically participating in it.

```
<beans>
  ...

  <bean id="myTxManager" class="org.springframework.orm.toplink.TopLinkTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
  </bean>

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

  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target" ref="myProductServiceTarget"/>
    <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>
```

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. Apache OJB

Apache OJB (http://db.apache.org/ojb) offers multiple API levels, such as ODMG and JDO. Aside from supporting OJB through JDO, Spring also supports OJB's lowe-level PersistenceBroker API as data access

strategy. The corresponding integration classes reside in the `org.springframework.orm.ojb` package.

12.5.1. OJB setup in a Spring environment

In contrast to Hibernate or JDO, OJB does not follow a factory object pattern for its resources. Instead, an OJB PersistenceBroker has to be obtained from the static PersistenceBrokerFactory class. That factory initializes itself from an OJB.properties file, residing in the root of the class path.

In addition to supporting OJB's default initialization style, Spring also provides a `Local`OjbConfigurer class that allows for using Spring-managed `DataSource` instances as OJB connection providers. The `DataSource` instances are referenced in the OJB repository descriptor (the mapping file), through the "jcd-alias" defined there: each such alias is matched against the Spring-managed bean of the same name.

```
<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="ojbConfigurer" class="org.springframework.orm.ojb.support.LocalOjbConfigurer"/>

  ...
</beans>
```

```
<descriptor-repository version="1.0">

  <jdbc-connection-descriptor jcd-alias="dataSource" default-connection="true" ...>
        ...
  </jdbc-connection-descriptor>

  ...
</descriptor-repository>
```

A PersistenceBroker can then be opened through standard OJB API, specifying a corresponding "PBKey", usually through the corresponding "jcd-alias" (or relying on the default connection).

12.5.2. PersistenceBrokerTemplate and PersistenceBrokerDaoSupport

Each OJB-based DAO will be configured with a "PBKey" through bean-style configuration, i.e. through a bean property setter. Such a DAO could be coded against plain OJB API, working with OJB's static `PersistenceBrokerFactory`, but will usually rather be used with Spring's `PersistenceBrokerTemplate`:

```
<beans>
  ...

  <bean id="myProductDao" class="product.ProductDaoImpl">
    <property name="jcdAlias" value="dataSource"/>  <!-- can be omitted (default) -->
  </bean>

</beans>
```

```
public class ProductDaoImpl implements ProductDao {

    private String jcdAlias;

    public void setJcdAlias(String jcdAlias) {
        this.jcdAlias = jcdAlias;
    }
```

```
    public Collection loadProductsByCategory(final String category) throws DataAccessException {
        PersistenceBrokerTemplate pbTemplate =
                new PersistenceBrokerTemplate(new PBKey(this.jcdAlias);
        return (Collection) pbTemplate.execute(new PersistenceBrokerCallback() {
            public Object doInPersistenceBroker(PersistenceBroker pb)
                    throws PersistenceBrokerException {

                Criteria criteria = new Criteria();
                criteria.addLike("category", category + "%");
                Query query = new QueryByCriteria(Product.class, criteria);

                List result = pb.getCollectionByQuery(query);
                // do some further stuff with the result list
                return result;
            }
        });
    }
}
```

A callback implementation can effectively be used for any OJB data access. `PersistenceBrokerTemplate` will ensure that `PersistenceBroker`s 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 `getObjectById`, `getObjectByQuery`, `store`, or `delete` call, `PersistenceBrokerTemplate` offers alternative convenience methods that can replace such one line callback implementations. Furthermore, Spring provides a convenient `PersistenceBrokerDaoSupport` base class that provides a `setJcdAlias` method for receiving an OJB JCD alias, and `getPersistenceBrokerTemplate` for use by subclasses. In combination, this allows for very simple DAO implementations for typical requirements:

```
public class ProductDaoImpl extends PersistenceBrokerDaoSupport implements ProductDao {

    public Collection loadProductsByCategory(String category) throws DataAccessException {
        Criteria criteria = new Criteria();
        criteria.addLike("category", category + "%");
        Query query = new QueryByCriteria(Product.class, criteria);

        return getPersistenceBrokerTemplate().getCollectionByQuery(query);
    }
}
```

As alternative to working with Spring's `PersistenceBrokerTemplate`, you can also code your OJB data access against plain OJB API, explictly opening and closing a `PersistenceBroker`. 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. `PersistenceBrokerDaoSupport` offers a variety of support methods for this scenario, for fetching and releasing a transactional `PersistenceBroker` as well as for converting exceptions.

12.5.3. Transaction management

To execute service operations within transactions, you can use Spring's common declarative transaction facilities. For example, you could define a `TransactionProxyFactoryBean` for a ProductService, which in turn delegates to the OJB-based ProductDao. Each specified method would then automatically get executed within a transaction, with all affected DAO operations automatically participating in it.

```
<beans>
  ...

  <bean id="myTxManager" class="org.springframework.orm.ojb.PersistenceBrokerTransactionManager">
    <property name="jcdAlias" value="dataSource"/>  <!-- can be omitted (default) -->
  </bean>
```

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

  <bean id="myProductService"
      class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="myTxManager"/>
    <property name="target" ref="myProductServiceTarget"/>
    <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>
```

Note that OJB's PersistenceBroker level does not track changes of loaded objects. Therefore, a PersistenceBroker transaction is essentially simply a database transaction at the PersistenceBroker level, just with an additional first-level cache for persistent objects. Lazy loading will work both with and without the PersistenceBroker being open, in contrast to Hibernate and JDO (where the original Session or PersistenceManager, respectively, needs to remain open).

`PersistenceBrokerTransactionManager` is capable of exposing an OJB transaction to JDBC access code that accesses the same JDBC `DataSource`. The `DataSource` to expose the transactions for needs to be specified explicitly; it won't be autodetected.

12.6. iBATIS SQL Maps

Through the `org.springframework.orm.ibatis` package, Spring supports iBATIS SQL Maps 1.x and 2.x (http://www.ibatis.com). The iBATIS support 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, for example through `TransactionProxyFactoryBean`. 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.

12.6.1. Overview and differences between iBATIS 1.x and 2.x

Spring supports both iBATIS SQL Maps 1.x and 2.x. First let's have a look at the differences between the two.

The XML config files have changed a bit, node and attribute names. Also the Spring classes you need to extend are different, as are some method names.

Table 12.1. iBATIS SQL Maps supporting classes for 1.x and 2.x

Feature	1.x	2.x
Creation of SqlMap(Client)	`SqlMapFactoryBean`	`SqlMapClientFactoryBean`
Template-style helper class	`SqlMapTemplate`	`SqlMapClientTemplate`
Callback to use MappedStatement	`SqlMapCallback`	`SqlMapClientCallback`

Feature	1.x	2.x
Super class for DAOs	`SqlMapDaoSupport`	`SqlMapClientDaoSupport`

12.6.2. iBATIS SQL Maps 1.x

12.6.2.1. Setting up the SqlMap

Using iBATIS SQL Maps involves creating SqlMap configuration files containing statements and result maps. Spring takes care of loading those using the `SqlMapFactoryBean`.

```
public class Account {

    private String name;
    private String email;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return this.email;
    }

    public void setEmail(String email) {
        this.email = email;
    }
}
```

Suppose we would want to map this class. We'd have to create the following `SqlMap`. Using the query, we can later on retrieve users through their email addresses. `Account.xml`:

```
<sql-map name="Account">

  <result-map name="result" class="examples.Account">
    <property name="name" column="NAME" columnIndex="1"/>
    <property name="email" column="EMAIL" columnIndex="2"/>
  </result-map>

  <mapped-statement name="getAccountByEmail" result-map="result">
    select ACCOUNT.NAME, ACCOUNT.EMAIL
    from ACCOUNT
    where ACCOUNT.EMAIL = #value#
  </mapped-statement>

  <mapped-statement name="insertAccount">
    insert into ACCOUNT (NAME, EMAIL) values (#name#, #email#)
  </mapped-statement>

</sql-map>
```

After having defined the Sql Map, we have to create a configuration file for iBATIS (`sqlmap-config.xml`):

```
<sql-map-config>

  <sql-map resource="example/Account.xml"/>

</sql-map-config>
```

iBATIS loads resources from the class path, so be sure to add the `Account.xml` file to the class path.

Using Spring, we can now very easily set up the `SqlMap`, using the `SqlMapFactoryBean`:

```
<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="sqlMap" class="org.springframework.orm.ibatis.SqlMapFactoryBean">
    <property name="configLocation" value="WEB-INF/sqlmap-config.xml"/>
  </bean>

  ...
</beans>
```

12.6.2.2. Using `SqlMapTemplate` and `SqlMapDaoSupport`

The `SqlMapDaoSupport` class offers a supporting class similar to the `HibernateDaoSupport` and the `JdoDaoSupport` classes. Let's implement a DAO:

```
public class SqlMapAccountDao extends SqlMapDaoSupport implements AccountDao {

    public Account getAccount(String email) throws DataAccessException {
        return (Account) getSqlMapTemplate().executeQueryForObject("getAccountByEmail", email);
    }

    public void insertAccount(Account account) throws DataAccessException {
        getSqlMapTemplate().executeUpdate("insertAccount", account);
    }
}
```

As you can see, we're using the pre-configured `SqlMapTemplate` to execute the query. Spring has initialized the `SqlMap` for us using the `SqlMapFactoryBean`, and when setting up the `SqlMapAccountDao` as follows, you're all set to go. Note that with iBATIS SQL Maps 1.x, the JDBC `DataSource` is usually specified on the DAO.

```
<beans>
  ...

  <bean id="accountDao" class="example.SqlMapAccountDao">
    <property name="dataSource" ref="dataSource"/>
    <property name="sqlMap" ref="sqlMap"/>
  </bean>

</beans>
```

Note that a `SqlMapTemplate` instance could also be created manually, passing in the `DataSource` and the `SqlMap` as constructor arguments. The `SqlMapDaoSupport` base class simply pre-initializes a `SqlMapTemplate` instance for us.

12.6.3. iBATIS SQL Maps 2.x

12.6.3.1. 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 changes a bit (`sqlmap-config.xml`):

```
<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 application context. 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>
```

12.6.3.2. 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.

12.6.3.3. 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>
```

Chapter 13. Web MVC framework

13.1. Introduction to the web MVC framework

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 FormController. This is a major difference to Struts.

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

Compared to WebWork, Spring has more differentiated object roles. It supports the notion of a Controller, an optional command or form object, and a model that gets passed to the view. The model will normally include the command or form object but also arbitrary reference data. Instead, a WebWork Action combines all those roles into one single object. WebWork does allow you to use existing business objects as part of your form, but 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, returning null as ModelAndView. In the normal case, a ModelAndView instance consists of a view name and a model Map, containing bean names and corresponding objects (like a command or form, containing reference data). View name resolution is highly configurable, either via bean names, via a properties file, or via your own ViewResolver implementation. The abstract model Map allows for complete abstraction of the view technology, without any hassle. 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 MVC

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 but powerful tag library that avoids HTML generation at any cost, allowing for maximum flexibility in terms of markup code.

13.2. The `DispatcherServlet`

Spring's web MVC framework is, like many other web MVC frameworks, a request-driven web MVC framework, designed around a 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 ApplicationContext and allows you to use every other feature Spring has.

Like ordinary servlets, the DispatcherServlet 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.

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

```
        <servlet-name>example</servlet-name>
        <url-pattern>*.form</url-pattern>
    </servlet-mapping>
</web-app>
```

In the example above, all requests ending with `.form` will be handled by the DispatcherServlet. The DispatcherServlet now needs to be configured.

As illustrated in Section 3.11, “Introduction to the ApplicationContext”, ApplicationContexts in Spring can be scoped. In the web MVC framework, each DispatcherServlet has its own `WebApplicationContext`, which inherits all the beans already defined in 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.

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).

The config location used by the DispatcherServlet can be modified through a servlet initialization parameter (see below for details).

The `WebApplicationContext` is just an ordinary 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 is knows which servlet it is associated with (by having a link to the `ServletContext`). The WebApplicationContext is bound in the ServletContext, and using `RequestContextUtils` you can always lookup the WebApplicationContext in case you need it.

The Spring DispatcherServlet has a couple of special beans it uses, in order to be able to process requests and render the appropriate views. 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, defaults are provided so you don't have to worry about configuring them.

Table 13.1. Special beans in the WebApplicationContext

Expression	Explanation
handler mapping(s)	(Section 13.4, “Handler mappings”) a list of pre- and postprocessors and controllers that will be executed if they match certain criteria (for instance a matching URL specified with the controller)
controller(s)	(Section 13.3, “Controllers”) the beans providing the actual functionality (or at least, access to the functionality) as part of the MVC triad
view resolver	(Section 13.5, “Views and resolving them”) capable of resolving view names to views, used by the DispatcherServlet
locale resolver	(Section 13.6, “Using locales”) capable of resolving the locale a client is using, in order to be able to offer internationalized views
theme resolver	(Section 13.7, “Using themes”) capable of resolving themes your web application can use, for example, to offer personalized layouts
multipart resolver	(Section 13.8, “Spring's multipart (fileupload) support”) offers functionality to process file uploads from HTML forms

Expression	Explanation
handlerexception resolver	(Section 13.9, “Handling exceptions”) offers functionality to map exceptions to views or implement other more complex exception handling code

When a DispatcherServlet is setup for use and a request comes in for that specific DispatcherServlet it starts processing it. The list below describes the complete process a request goes through if 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 and if they are found, it is wrapped in a `MultipartHttpServletRequest` for further processing by other elements in the process. (See 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, controllers) will be executed in order to prepare a model.
6. If a model is returned, the view is rendered, using the view resolver that has been configured with the WebApplicationContext. If no model 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 might be thrown during processing of the request get picked up by any of the handlerexception resolvers that are declared in the WebApplicationContext. Using these exception resolvers you can define custom behavior in case such exceptions get thrown.

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

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

Table 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).

Parameter	Explanation
namespace	the namespace of the `WebApplicationContext`. Defaults to `[server-name]-servlet`.

13.3. Controllers

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

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

```
public interface Controller {

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

As you can see, the `Controller` interface requires a single method that should be capable of 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 controllers that already contain a lot of the functionality you might need. The `Controller` interface just defines the most common functionality required of every controller - handling a request and returning a model and a view.

13.3.1. AbstractController and WebContentGenerator

Of course, just a controller interface isn't enough. To provide a basic infrastructure, all of Spring's Controllers 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 (using a `ServletException)`).
`requiresSession`	indicates whether or not this controller requires a session to do its work. This feature is offered to all controllers. If a session is not present when such a controller receives a request, the user is informed using a `ServletException`.
`synchronizeSession`	use this if you want handling by this controller to be synchronized on the user's session. To be more specific, extending controller will override the `handleRequestInternal` method, which will be synchronized if you specify this

Feature	Explanation
	variable.
`cacheSeconds`	when you want a controller to generate a caching directive in the HTTP response, specify a positive integer here. By default it is set to *-1* so no caching directives will be included.
`useExpiresHeader`	tweaks your controllers to specify the HTTP 1.0 compatible *"Expires"* header. By default it's set to true, so you won't have to change it.
`useCacheHeader`	tweaks your controllers to specify the HTTP 1.1 compatible *"Cache-Control"* header. By default this is set to true so you won't have to change it.

The last two properties are actually part of the `WebContentGenerator` 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 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("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 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 returns an hard-coded view (hmm, not so nice), named index (see Section 13.5, “Views and resolving them” for more information about views).

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 (ahhh, no need to hard-code the viewname).

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

13.3.3. The `MultiActionController`

Spring offers a multi-action controller with which you aggregate multiple actions into one controller, grouping functionality together. The multi-action controller lives in a separate package - `org.springframework.web.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 Resolver will be invoked. If you choose this scenario, you will have to define the delegate using this configuration parameter as a collaborator.
`methodNameResolver`	somehow the MultiActionController will need to resolve the method it has to invoke, based on the request that came in. You can define a resolver that is capable of doing that using this configuration parameter.

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

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

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

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

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

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

- `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` configuration parameter specifies the parameter that is inspected).
- `InternalPathMethodNameResolver` - retrieves the filename from the path and uses that as the method name (`http://www.sf.net/testing.view` will result in a method `testing(HttpServletRequest, HttpServletResponse)` being called).
- `PropertiesMethodNameResolver` - uses a user-defined properties object with request URLs mapped to methodnames. When the properties contain `/index/welcome.html=doIt` and a request to `/index/welcome.html` comes in, the `doIt(HttpServletRequest, HttpServletResponse)` method is called. This method name resolver works with the `PathMatcher`, 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</value></property>
</bean>

<bean id="paramMultiController" class="org....mvc.multiaction.MultiActionController">
  <property name="methodNameResolver"><ref bean="paramResolver"/></property>
  <property name="delegate"><ref bean="sampleDelegate"/></property>
</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">
    <props>
      <prop key="/index/welcome.html">retrieveIndex</prop>
      <prop key="/**/notwelcome.html">retrieveIndex</prop>
      <prop key="/*/user?.html">retrieveIndex</prop>
    </props>
  </property>
</bean>

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

13.3.4. CommandControllers

Spring's *CommandControllers* are a fundamental part of the Spring 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 similar role to Struts' ActionForm, but in Spring, your data objects don't have to implement a framework-specific interface. First, let's examine what command controllers available, to get 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, the 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 FormController 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 WizardController 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. `HandlerInterceptor`s are discussed in Section 13.4.3,

"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.
- `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 FormController (see Section 13.3.4, "CommandControllers" for more information on Command- and FormControllers) and a JSP view (or Velocity template) that renders the form. When using the BeanNameUrlHandlerMapping, we could map the HTTP request with URL `http://samples.com/editaccount.form` to the appropriate FormController as follows:

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

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

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

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

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

NOTE: if you want to use the `BeanNameUrlHandlerMapping`, you don't necessarily have to define it in the 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 `org.springframework.util.PathMatcher`). 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>
    <servlet-mapping>
        <servlet-name>sample</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
    ...
</web-app>
```

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

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

    <bean id="someViewController"
          class="org.springframework.web.servlet.mvc.UrlFilenameViewController"/>

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

This handler mapping routes requests for `help.html` in any directory to the `helpController`, which is a UrlFilenameViewController (more about controllers can be found in Section 13.3, “Controllers”). Requests for a resource beginning with `view`, and ending with `.html` in the directory `ex`, will be routed to the `someViewController`. Two further mappings are defined for `editAccountFormController`.

13.4.3. Adding `HandlerInterceptors`

Spring's handler mapping mechanism has a 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">
            <props>
                <prop key="/*.form">editAccountFormController</prop>
                <prop key="/*.view">editAccountFormController</prop>
            </props>
        </property>
    </bean>

    <bean id="officeHoursInterceptor"
          class="samples.TimeBasedAccessInterceptor">
        <property name="openingTime"><value>9</value></property>
        <property name="closingTime"><value>18</value></property>
    </bean>
<beans>
```

```
package samples;

public class TimeBasedAccessInterceptor extends HandlerInterceptorAdapter {

    private int openingTime;
    private int closingTime;
    public void setOpeningTime(int openingTime) {
        this.openingTime = openingTime;
    }
    public void setClosingTime(int closingTime) {
        this.closingTime = closingTime;
    }
    public boolean preHandle(
            HttpServletRequest request,
            HttpServletResponse response,
            Object handler)
    throws Exception {
        Calendar cal = Calendar.getInstance();
        int hour = cal.get(HOUR_OF_DAY);
        if (openingTime <= hour < closingTime) {
            return true;
        } else {
            response.sendRedirect("http://host.com/outsideOfficeHours.html");
            return false;
        }
    }
}
```

Any request coming in, will be intercepted by the `TimeBasedAccessInterceptor`, and if the current time is outside office hours, the user will be 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 to make it easy for you to extend the `HandlerInterceptor`.

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 Java Server Pages, Velocity templates and XSLT views, for example. Chapter 14, *Integrating view technologies* has details of integrating various 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. ViewResolvers

As discussed in 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 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`.
UrlBasedViewResolver	A simple implementation of ViewResolver that allows for 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 like JstlView and TilesView. The view class for all views generated by this resolver can be specified via setViewClass. See UrlBasedViewResolver's javadocs for details.
VelocityViewResolver / 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/</value></property>
    <property name="suffix"><value>.jsp</value></property>
</bean>
```

When returning `test` as a viewname, this view resolver will hand the request over to the RequestDispatcher that 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</value></property>
    <property name="defaultParentView"><value>parentView</value></property>
</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 technology. 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>

### 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 tehcnologies 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 desireable to issue an HTTP redirect back to the client, before the view is rendered. This si desireable for example when one controller has been called with POSTed 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, whih 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 possiblity 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 it's 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 generally always preferable if the redirect URL at least is injected into the Controller, so that it is not baked into the controller but rather configured in the context along with view names and the like.

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 achived. 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 primarilly using another view technology, but want to still be able to in some cases force a forward to happen to a resource to be handled by the Servlet/JSP engine. Note that if you need to do this a lot though, you may also just chain multiple view resolvers.

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, “Adding HandlerInterceptors” 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.

```
<bean id="localeResolver">
    <property name="cookieName"><value>clientlanguage</value></property>
    <!-- in seconds. If set to -1, the cookie is not persisted (deleted when browser shuts down) -->
    <property name="cookieMaxAge"><value>100000</value></property>
</bean>
```

This is an example of defining a CookieLocaleResolver.

Table 13.6. Special beans in the WebApplicationContext

Property	Default	Description
cookieName	classname + LOCALE	The name of the cookie
cookieMaxAge	Integer.MAX_INT	The maximum time a cookie will stay persistent on the client. If -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</value></property>
</bean>

<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver"/>

<bean id="urlMapping"
      class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="localeChangeInterceptor"/>
        </list>
    </property>
    <property name="mappings">
        <props>
            <prop key="/**/*.view">someController</prop>
        </props>
    </property>
```

```
</bean>
```

All calls to all *.view resources containing a parameter named `siteLanguage` will now change the locale. So a call to `http://www.sf.net/home.view?siteLanguage=nl` will change the site language to Dutch.

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 setup a `org.springframework.ui.context.ThemeSource`. The `WebApplicationContext` interface extends `ThemeSource` but delegates its responsabilities 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's 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 internationalisation 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 `LocalResolver`. 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 client'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</value>
    </property>
</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</value>
    </property>
</bean>
```

Of course you need to stick 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`, while 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 Multipart request, it activates the resolver that has been declared in your context and hands over the request. What it basically does is wrap the current `HttpServletRequest` into a `MultipartHttpServletRequest` that has support for multiparts. Using the MultipartHttpServletRequest you can get information about the multiparts contained by this request and actually get the multiparts themselves in your controllers.

13.8.3. Handling a fileupload 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, then let Spring bind the file on your form. Just as with any other property that's not automagically convertible to a String or primitive type, to be able to put binary data in your beans you have to register a custom editor with the `ServletRequestDatabinder`. There are a couple of editors available for handling files and setting the results on a bean. 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 form in a website, declare the resolver, a url mapping to a controller that will process the bean, and the controller itself.

```
<beans>

    ...

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

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

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

</beans>
```

After that, create the controller and the actual bean to hold the file property

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

    protected ModelAndView onSubmit(
        HttpServletRequest request,
        HttpServletResponse response,
        Object command,
```

```
        BindException errors)
        throws ServletException, IOException {

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

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

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

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

}

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

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

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

As you can see, the FileUploadBean has a property typed byte[] that holds the file. The controller 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 these examples, 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).

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

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

As you can see, we've created a field named after the property of the bean that holds the byte[]. Furthermore we've added the encoding attribute which is necessary to let the browser know how to encode the multipart fields (do not forget this!). Now everything should work.

13.9. 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`, 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.

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 viewresolver defined in the WebApplicationContext. Furthermore, of course you need to write some JSPs that will actually render the view. This part describes some of the additional features Spring provides to facilitate JSP development.

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 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 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. More information about the individual tags can be found online: http://www.springframework.org/docs/taglib/index.html.

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.

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 Lang`
- `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://jakarta.apache.org/struts). 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="factoryClass" value="org.apache.struts.tiles.xmlDefinition.I18nFactorySet"/>
  <property name="definitions">
    <list>
      <value>/WEB-INF/defs/general.xml</value>
      <value>/WEB-INF/defs/widgets.xml</value>
      <value>/WEB-INF/defs/administrator.xml</value>
      <value>/WEB-INF/defs/customer.xml</value>
      <value>/WEB-INF/defs/templates.xml</value>
    </list>
  </property>
</bean>
```

As you can see, there are five files containing definitions, which are all located in the WEB-INF/defs directory.

At initialization of the WebApplicationContext, the files will be loaded and the definitionsfactory defined by the `factoryClass`-property is initialized. After that has been done, the tiles includes in the definition files can be used as views within your Spring 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 `InternalResourceViewResolver` and the `ResourceBundleViewResolver`.

14.3.2.1. InternalResourceViewResolver

The InternalResourceViewResolver instantiates the given `viewClass` for each view it has to resolve.

```
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="requestContextAttribute" value="requestContext"/>
  <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 (<b>this is the name of a definition</b>)

vetsView.class=org.springframework.web.servlet.view.tiles.TilesView
vetsView.url=vetsView (<b>again, this is the name of a definition</b>)

findOwnersForm.class=org.springframework.web.servlet.view.JstlView
findOwnersForm.url=/WEB-INF/jsp/findOwners.jsp
...
```

As you can see, when using the ResourceBundleViewResolver, you can mix view using different view technologies.

14.4. Velocity & FreeMarker

Velocity [http://jakarta.apache.org/velocity] and FreeMarker [http://www.freemarker.org] 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 [http://www.freemarker.org] 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` folder 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"/>
</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"/>
</bean>
```

NB: 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 in Section 14.4.2, “Context configuration” 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 [http://www.springframework.org/docs/api/org/springframework/ui/velocity/VelocityEngineFactory.html] 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" 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` . The same property can be set 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
```

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 $default)`	`<@spring.messageText code, default/>`
url (prefix a relative URL with the application's context root)	`#springUrl($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/>`

macro	VTL definition	FTL definition
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 $attributes)`	`<@spring.formCheckboxes path, options, separator, attributes/>`
showErrors (simplify display of validation errors for the bound field)	`#springShowErrors($separator $classOrStyle)`	`<@spring.showErrors separator, classOrStyle/>`

* 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 "<br>").
- 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. Overriding HTML escaping and making tags XHTML compliant

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 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 Section 13.3, “Controllers” for details of Spring `Controller`s. 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 implements 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 req,
    HttpServletResponse resp)
    throws Exception {

    Map map = new HashMap();
    List wordList = new ArrayList();
```

```
    wordList.add("hello");
    wordList.add("world");

    map.put("wordList", wordList);

    return new ModelAndView("home", map);
}
```

So far we've done nothing that's XSLT specific. The model data has been created in the same way as you would for any other Spring MVC application. Depending on the configuration of the application now, that list of words could be rendered by JSP/JSTL by having them added as request attributes, or they could be handled by Velocity by adding the object to the VelocityContext. In order to have XSLT render them, they of course have to be converted into an XML document somehow. There are software packages available that will automatically 'domify' an object graph, but within Spring, you have complete flexibility to create the DOM from your model in any way you choose. This prevents the transformation of XML playing too great a part in the structure of your model data which is a danger when using tools to manage the domification process.

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 subclass `org.springframework.web.servlet.view.xslt.AbstractXsltView`. In doing so, we must implement the abstract method `createDomNode()`. The first parameter passed to this method is our model Map. Here's the complete listing of the HomePage class in our trivial word application - it uses JDOM to build the XML document before converting it to the required W3C Node, but this is simply because I find JDOM (and Dom4J) easier API's to handle than the W3C API.

```
package xslt;

// imports omitted for brevity

public class HomePage extends AbstractXsltView {

    protected Node createDomNode(
        Map model, String rootName, HttpServletRequest req, HttpServletResponse res
    ) throws Exception {

        org.jdom.Document doc = new org.jdom.Document();
        Element root = new Element(rootName);
        doc.setRootElement(root);

        List words = (List) model.get("wordList");
        for (Iterator it = words.iterator(); it.hasNext();) {
            String nextWord = (String) it.next();
            Element e = new Element("word");
            e.setText(nextWord);
            root.addContent(e);
        }

        // convert JDOM doc to a W3C Node and return
        return new DOMOutputter().output( doc );
    }

}
```

14.5.1.3.1. Adding stylesheet parameters

A series of parameter name/value pairs can optionally be defined by your subclass which will be added to the transformation object. The parameter names must match those defined in your XSLT template declared with `<xsl:param name="myParam">defaultValue</xsl:param>` To specify the parameters, override the method `getParameters()` from AbstractXsltView and return a `Map` of the name/value pairs. If your parameters need to derive information from the current request, you can (from version 1.1) override the

`getParameters(HttpServletRequest request)` method instead.

14.5.1.3.2. Formatting dates and currency

Unlike JSTL and Velocity, XSLT has relatively poor support for locale based currency and date formatting. In recognition of the fact, Spring provides a helper class that you can use from within your `createDomNode()` methods to get such support. See the javadocs for `org.springframework.web.servlet.view.xslt.FormatHelper`

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 obviously points to the XSLT file which will handle the XML transformation into HTML for us and the final property '.root' is the name that will be used as the root of the XML document. This gets passed to the HomePage class above in the second parameter to the `createDomNode` method.

14.5.1.5. Document transformation

Finally, we have the XSLT code used for transforming the above document. As highlighted in the views.properties file, it is called `home.xslt` and it lives in the war file under `WEB-INF/xsl`.

```
<?xml version="1.0"?>

<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
    <xsl:output method="text/html" omit-xml-declaration="yes"/>

    <xsl:template match="/">
        <html>
            <head><title>Hello!</title></head>
            <body>

                <h1>My First Words</h1>
                <xsl:for-each select="wordList/word">
                    <xsl:value-of select="."/><br />
                </xsl:for-each>

            </body>
        </html>
    </xsl:template>

</xsl:stylesheet>
```

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 formats. 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 `ApplicationContext` you have 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 `ResourceBundleViewResolver` which will look for view mappings in the resource bundle with base name `views`. The exact contents of this file is described in the next section.

14.7.2.2. Configuring the `View`s

Spring contains five different View implementations for JasperReports four of which corresponds 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`	Decided at runtime (see Section 14.7.2.4, “Using JasperReportsMultiFormatView”)

Mapping one of these classes to a view name and a report file is simply 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. This will cause 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 Spring you can map either of these files to your report file and Spring will take care of compiling the `.jrxml` file on the fly for you. You should note that after a `.jrxml` file is compiled by Spring, the compiled report is cached for the life 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 `Collection` 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 this:

```
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 this:

```
<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

Spring can be easily integrated into any Java-based web framework. All you need to do is to declare the `ContextLoaderListener` [http://www.springframework.org/docs/api/org/springframework/web/context/ContextLoaderListener.html] in your `web.xml` and use a *contextConfigLocation* <context-param> to set which context files to load.

The <context-param>:

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

The <listener>:

```
<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` [http://www.springframework.org/docs/api/org/springframework/web/context/ContextLoaderServlet.html] to achieve this same functionality.

If you don't specify the *contextConfigLocation* context parameter, the `ContextLoaderListener` will look for a */WEB-INF/applicationContext.xml* file to load. Once the context files are loaded, Spring creates a `WebApplicationContext` [http://www.springframework.org/docs/api/org/springframework/web/context/WebApplicationContext.html] object based on the bean definitions and puts it into the `ServletContext`.

All Java web frameworks are built on top of the Servlet API, so you can use the following code to get the `ApplicationContext` that Spring created.

```
WebApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(servletContext);
```

The `WebApplicationContextUtils` [http://www.springframework.org/docs/api/org/springframework/web/context/support/WebApplicationContextUtils.html] 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 the `BeanFactory`, but they also allow you to use dependency injection on their controllers. Each framework section has more detail on its specific integration strategies.

15.2. JavaServer Faces

JavaServer Faces (JSF) is a component-based, event-driven web framework. According to Sun Microsystem's JSF Overview [http://java.sun.com/j2ee/javaserverfaces/overview.html], JSF technology includes:

- A set of APIs for representing UI components and managing their state, handling events and input validation, defining page navigation, and supporting internationalization and accessibility.
- A JavaServer Pages (JSP) custom tag library for expressing a JavaServer Faces interface within a JSP page.

15.2.1. DelegatingVariableResolver

The easiest way to integrate your Spring middle-tier with your JSF web layer is to use the `DelegatingVariableResolver` [http://www.springframework.org/docs/api/org/springframework/web/jsf/DelegatingVariableResolver.html] class. To configure this variable resolver in your application, you'll need to edit your *faces-context.xml*. 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`:

```
<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>
```

By specifying Spring's variable resolver, you can configure Spring beans as managed properties of your managed beans. The `DelegatingVariableResolver` will first delegate value lookups to the default resolver of the underlying JSF implementation, and then to Spring's root `WebApplicationContext`. This allows you to easily inject dependencies into your JSF-managed beans.

Managed beans are defined in your `faces-config.xml` file. Below is an example where #{userManager} is a bean that's retrieved from Spring's `BeanFactory`.

```
<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>
```

The `DelegatingVariableResolver` is the recommended strategy for integrating JSF and Spring. If you're looking for more robust integration features, you might take a look at the JSF-Spring [http://jsf-spring.sourceforge.net/] project.

15.2.2. FacesContextUtils

A custom VariableResolver works well when mapping your properties to beans in *faces-config.xml*, but at times you may need to grab a bean explicitly. The `FacesContextUtils`

[http://www.springframework.org/docs/api/org/springframework/web/jsf/FacesContextUtils.html] class makes this easy. It's similar to `WebApplicationContextUtils`, except that it takes a `FacesContext` parameter rather than a `ServletContext` parameter.

```
ApplicationContext ctx = FacesContextUtils.getWebApplicationContext(FacesContext.getCurrentInstance());
```

15.3. Struts

Struts [http://struts.apache.org] 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 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.

15.3.1. ContextLoaderPlugin

The `ContextLoaderPlugin` [http://www.springframework.org/docs/api/org/springframework/web/struts/ContextLoaderPlugIn.html] 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.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 [http://sourceforge.net/tracker/index.php?func=detail&aid=1088866&group_id=39190&atid=424562] for this issue.

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-context.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" .../>
```

15.3.1.1. DelegatingRequestProcessor

To configure the `DelegatingRequestProcessor` [http://www.springframework.org/docs/api/org/springframework/web/struts/DelegatingRequestProcessor.html] 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're using Tiles in your Struts application, you must configure your <controller> with the `DelegatingTilesRequestProcessor` [http://www.springframework.org/docs/api/org/springframework/web/struts/DelegatingTilesRequestProcessor.html].

15.3.1.2. DelegatingActionProxy

If you have a custom `RequestProcessor` and can't use the `DelegatingTilesRequestProcessor`, you can use the `DelegatingActionProxy` [http://www.springframework.org/docs/api/org/springframework/web/struts/DelegatingActionProxy.html] 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 *singleton="false"* to your Action's bean definition.

```
<bean name="/user" singleton="false" autowire="byName"
    class="org.example.web.UserAction"/>
```

15.3.2. 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` [http://www.springframework.org/docs/api/org/springframework/web/struts/ActionSupport.html] 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` [http://www.springframework.org/docs/api/org/springframework/web/struts/ActionSupport.html],
- `DispatchActionSupport` [http://www.springframework.org/docs/api/org/springframework/web/struts/DispatchActionSupport.html],
- `LookupDispatchActionSupport` [http://www.springframework.org/docs/api/org/springframework/web/struts/LookupDispatchActionSupport.html] and
- `MappingDispatchActionSupport` [http://www.springframework.org/docs/api/org/springframework/web/struts/MappingDispatchActionSupport.html].

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.

15.4. Tapestry

Tapestry is a powerful, component-oriented web application framework from Apache's Jakarta project (http://jakarta.apache.org/tapestry). While Spring has its own powerful web ui layer, there are a number of unique advantages to building a J2EE application using a combination of Tapestry for the web ui, and the Spring container for the lower layers. This document attempts to detail a few best practices for combining these two frameworks. It is expected that you are relatively familiar with both Tapestry and Spring Framework basics, so they will not be explained here. General introductory documentation for both Tapestry and Spring Framework are available on their respective web sites.

15.4.1. Architecture

A typical layered J2EE application built with Tapestry and Spring will consist of a top UI layer built with Tapestry, and a number of lower layers, hosted out of one or more Spring Application Contexts.

- *User Interface Layer:*

  - concerned with the user interface

  - contains some application logic

  - provided by Tapestry

  - aside from providing UI via Tapestry, code in this layer does its work via objects which implement interfaces from the Service Layer. The actual objects which implement these service layer interfaces are obtained from a Spring Application Context.

- *Service Layer:*

  - application specific 'service' code

  - works with domain objects, and uses the Mapper API to get those domain objects into and out of some sort of repository (database)

  - hosted in one or more Spring contexts

  - code in this layer manipulates objects in the domain model, in an application specific fashion. It does its work via other code in this layer, and via the Mapper API. An object in this layer is given the specific mapper implementations it needs to work with, via the Spring context.

  - since code in this layer is hosted in the Spring context, it may be transactionally wrapped by the Spring context, as opposed to managing its own transactions

- *Domain Model:*

  - domain specific object hierarchy, which deals with data and logic specific to this domain

  - although the domain object hierarchy is built with the idea that it is persisted somehow and makes some general concessions to this (for example, bidirectional relationships), it generally has no knowledge of other layers. As such, it may be tested in isolation, and used with different mapping implementations for production vs. testing.

  - these objects may be standalone, or used in conjunction with a Spring application context to take advantage of some of the benefits of the context, e.g., isolation, inversion of control, different strategy implementations, etc.

- *Data Source Layer:*

  - Mapper API (also called Data Access Objects): an API used to persist the domain model to a repository of some sort (generally a DB, but could be the filesystem, memory, etc.)

  - Mapper API implementations: one or more specific implementations of the Mapper API, for example, a Hibernate-specific mapper, a JDO-specific mapper, JDBC-specific mapper, or a memory mapper.

  - mapper implementations live in one or more Spring Application Contexts. A service layer object is given the mapper objects it needs to work with via the context.

- *Database, filesystem, or other repositories:*

  - objects in the domain model are stored into one or more repositories via one or more mapper implementations

  - a repository may be very simple (e.g. filesystem), or may have its own representation of the data from the domain model (i.e. a schema in a db). It does not know about other layers howerver.

15.4.2. Implementation

The only real question (which needs to be addressed by this document), is how Tapestry pages get access to service implementations, which are simply beans defined in an instance of the Spring Application Context.

15.4.2.1. Sample application context

Assume we have the following simple Application Context definition, in xml form:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
        "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <!-- ========================= GENERAL DEFINITIONS ========================= -->

    <!-- ========================= PERSISTENCE DEFINITIONS ========================= -->

    <!-- the DataSource -->
    <bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
        <property name="jndiName"><value>java:DefaultDS</value></property>
        <property name="resourceRef"><value>false</value></property>
    </bean>

    <!-- define a Hibernate Session factory via a Spring LocalSessionFactoryBean -->
    <bean id="hibSessionFactory"
        class="org.springframework.orm.hibernate.LocalSessionFactoryBean">
        <property name="dataSource"><ref bean="dataSource"/></property>
    </bean>

    <!--
     - Defines a transaction manager for usage in business or data access objects.
     - No special treatment by the context, just a bean instance available as reference
     - for business objects that want to handle transactions, e.g. via TransactionTemplate.
     -->
    <bean id="transactionManager"
        class="org.springframework.transaction.jta.JtaTransactionManager">
    </bean>

    <bean id="mapper"
        class="com.whatever.dataaccess.mapper.hibernate.MapperImpl">
        <property name="sessionFactory"><ref bean="hibSessionFactory"/></property>
    </bean>

    <!-- ========================= BUSINESS DEFINITIONS ========================= -->
```

```
    <!-- AuthenticationService, including tx interceptor -->
    <bean id="authenticationServiceTarget"
        class="com.whatever.services.service.user.AuthenticationServiceImpl">
        <property name="mapper"><ref bean="mapper"/></property>
    </bean>
    <bean id="authenticationService"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager"><ref bean="transactionManager"/></property>
        <property name="target"><ref bean="authenticationServiceTarget"/></property>
        <property name="proxyInterfacesOnly"><value>true</value></property>
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

    <!-- UserService, including tx interceptor -->
    <bean id="userServiceTarget"
        class="com.whatever.services.service.user.UserServiceImpl">
        <property name="mapper"><ref bean="mapper"/></property>
    </bean>
    <bean id="userService"
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager"><ref bean="transactionManager"/></property>
        <property name="target"><ref bean="userServiceTarget"/></property>
        <property name="proxyInterfacesOnly"><value>true</value></property>
        <property name="transactionAttributes">
            <props>
                <prop key="*">PROPAGATION_REQUIRED</prop>
            </props>
        </property>
    </bean>

</beans>
```

Inside the Tapestry application, we need to load this application context, and allow Tapestry pages to get the authenticationService and userService beans, which implement the AuthenticationService and UserService interfaces, respectively.

15.4.2.2. Obtaining beans in Tapestry pages

At this point, the application context is available to a web application by calling Spring's static utility function `WebApplicationContextUtils.getApplicationContext(servletContext)`, where servletContext is the standard `ServletContext` from the J2EE Servlet specification. As such, one simple mechanism for a page to get an instance of the UserService, for example, would be with code such as:

```
    WebApplicationContext appContext = WebApplicationContextUtils.getApplicationContext(
        getRequestCycle().getRequestContext().getServlet().getServletContext());
    UserService userService = (UserService) appContext.getBean("userService");
    ... some code which uses UserService
```

This mechanism does work. It can be made a lot less verbose by encapsulating most of the functionality in a method in the base class for the page or component. However, in some respects it goes against the Inversion of Control approach which Spring encourages, which is being used in other layers of this app, in that ideally you would like the page to not have to ask the context for a specific bean by name, and in fact, the page would ideally not know about the context at all.

Luckily, there is a mechanism to allow this. We rely upon the fact that Tapestry already has a mechanism to declaratively add properties to a page, and it is in fact the preferred approach to manage all properties on a page in this declarative fashion, so that Tapestry can properly manage their lifecycle as part of the page and component lifecycle.

15.4.2.3. Exposing the application context to Tapestry

First we need to make the `ApplicationContext` available to the Tapestry page or Component without having to have the `ServletContext`; this is because at the stage in the page's/component's lifecycle when we need to access the `ApplicationContext`, the `ServletContext` won't be easily available to the page, so we can't use `WebApplicationContextUtils.getApplicationContext(servletContext)` directly. One way is by defining a custom version of the Tapestry IEngine which exposes this for us:

```
package com.whatever.web.xportal;
...
import ...
...
public class MyEngine extends org.apache.tapestry.engine.BaseEngine {

    public static final String APPLICATION_CONTEXT_KEY = "appContext";

    /**
     * @see org.apache.tapestry.engine.AbstractEngine#setupForRequest(org.apache.tapestry.request.RequestContext
     */
    protected void setupForRequest(RequestContext context) {
        super.setupForRequest(context);

        // insert ApplicationContext in global, if not there
        Map global = (Map) getGlobal();
        ApplicationContext ac = (ApplicationContext) global.get(APPLICATION_CONTEXT_KEY);
        if (ac == null) {
            ac = WebApplicationContextUtils.getWebApplicationContext(
                context.getServlet().getServletContext()
            );
            global.put(APPLICATION_CONTEXT_KEY, ac);
        }
    }
}
```

This engine class places the Spring Application Context as an attribute called "appContext" in this Tapestry app's 'Global' object. Make sure to register the fact that this special IEngine instance should be used for this Tapestry application, with an entry in the Tapestry application definition file. For example:

```
file: xportal.application:

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE application PUBLIC
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">
<application
    name="Whatever xPortal"
    engine-class="com.whatever.web.xportal.MyEngine">
</application>
```

15.4.2.4. Component definition files

Now in our page or component definition file (*.page or *.jwc), we simply add property-specification elements to grab the beans we need out of the ApplicationContext, and create page or component properties for them. For example:

```
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>
```

The OGNL expression inside the property-specification specifies the initial value for the property, as a bean obtained from the context. The entire page definition might look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE page-specification PUBLIC
```

```
    "-//Apache Software Foundation//Tapestry Specification 3.0//EN"
    "http://jakarta.apache.org/tapestry/dtd/Tapestry_3_0.dtd">

<page-specification class="com.whatever.web.xportal.pages.Login">

    <property-specification name="username" type="java.lang.String"/>
    <property-specification name="password" type="java.lang.String"/>
    <property-specification name="error" type="java.lang.String"/>
    <property-specification name="callback" type="org.apache.tapestry.callback.ICallback" persistent="yes"/>
    <property-specification name="userService"
                            type="com.whatever.services.service.user.UserService">
        global.appContext.getBean("userService")
    </property-specification>
    <property-specification name="authenticationService"
                            type="com.whatever.services.service.user.AuthenticationService">
        global.appContext.getBean("authenticationService")
    </property-specification>

    <bean name="delegate" class="com.whatever.web.xportal.PortalValidationDelegate"/>

    <bean name="validator" class="org.apache.tapestry.valid.StringValidator" lifecycle="page">
        <set-property name="required" expression="true"/>
        <set-property name="clientScriptingEnabled" expression="true"/>
    </bean>

    <component id="inputUsername" type="ValidField">
        <static-binding name="displayName" value="Username"/>
        <binding name="value" expression="username"/>
        <binding name="validator" expression="beans.validator"/>
    </component>

    <component id="inputPassword" type="ValidField">
        <binding name="value" expression="password"/>
       <binding name="validator" expression="beans.validator"/>
       <static-binding name="displayName" value="Password"/>
       <binding name="hidden" expression="true"/>
    </component>

</page-specification>
```

15.4.2.5. Adding abstract accessors

Now in the Java class definition for the page or component itself, all we need to do is add an abstract getter method for the properties we have defined, to access them. When the page or component is actually loaded by Tapestry, it performs runtime code instrumentation on the classfile to add the properties which have been defined, and hook up the abstract getter methods to the newly created fields. For example:

```
    // our UserService implementation; will come from page definition
    public abstract UserService getUserService();
    // our AuthenticationService implementation; will come from page definition
    public abstract AuthenticationService getAuthenticationService();
```

For completeness, the entire Java class, for a login page in this example, might look like this:

```
package com.whatever.web.xportal.pages;

/**
 *  Allows the user to login, by providing username and password.
 *  After 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 a cookie to store on the user's machine that will identify
     * them next time they log in.
     **/
    private static final String COOKIE_NAME = Login.class.getName() + ".username";
```

```
    private final static int ONE_WEEK = 7 * 24 * 60 * 60;

    // --- attributes

    public abstract String getUsername();
    public abstract void setUsername(String username);

    public abstract String getPassword();
    public abstract void setPassword(String password);

    public abstract ICallback getCallback();
    public abstract void setCallback(ICallback value);

    public abstract UserService getUserService();

    public abstract AuthenticationService getAuthenticationService();

    // --- methods

    protected IValidationDelegate getValidationDelegate() {
        return (IValidationDelegate) getBeans().getBean("delegate");
    }

    protected void setErrorField(String componentId, String message) {
        IFormComponent field = (IFormComponent) getComponent(componentId);
        IValidationDelegate delegate = getValidationDelegate();
        delegate.setFormComponent(field);
        delegate.record(new ValidatorException(message));
    }

    /**
     *  Attempts to login.
     *
     *  <p>If the user name is not known, or the password is invalid, then an error
     *  message is displayed.
     *
     **/
    public void attemptLogin(IRequestCycle cycle) {

        String password = getPassword();

        // Do a little extra work to clear out the password.

        setPassword(null);
        IValidationDelegate delegate = getValidationDelegate();

        delegate.setFormComponent((IFormComponent) getComponent("inputPassword"));
        delegate.recordFieldInputValue(null);

        // An error, from a validation field, may already have 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);

        // I've found that failing to set a maximum age and a path means that
        // the browser (IE 5.0 anyway) quietly drops the cookie.

        IEngine engine = getEngine();
        Cookie cookie = new Cookie(COOKIE_NAME, username);
        cookie.setPath(engine.getServletPath());
        cookie.setMaxAge(ONE_WEEK);

        // Record the user's username in a cookie

        cycle.getRequestContext().addCookie(cookie);

        engine.forgetPage(getPageName());
    }

    public void pageBeginRender(PageEvent event) {
        if (getUsername() == null)
            setUsername(getRequestCycle().getRequestContext().getCookieValue(COOKIE_NAME));
    }
}
```

15.4.3. Summary

In this example, we've managed to allow service beans defined in the Spring `ApplicationContext` to be provided to the page in a declarative fashion. The page class does not know where the service implementations are coming from, and in fact it is easy to slip in another implementation, for example, during testing. This inversion of control is one of the prime goals and benefits of the Spring Framework, and we have managed to extend it all the way up the J2EE stack in this Tapestry application.

15.5. WebWork

WebWork [http://www.opensymphony.com/webwork] is a web framework designed with simplicity in mind. It's built on top of XWork [http://www.opensymphony.com/xwork], which is a generic command framework. XWork also has an IoC container, but it isn't as full-featured as Spring and won't be covered in this section. WebWork controllers are called Actions, mainly because they must implement the `Action` [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/Action.html] interface. The `ActionSupport` [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/ActionSupport.html] class implements this interface, and it is most common parent class for WebWork actions.

WebWork maintains its own Spring integration project, located on java.net in the xwork-optional [https://xwork-optional.dev.java.net/] project. Currently, three options are available for integrating WebWork with Spring:

- **SpringObjectFactory**: override XWork's default `ObjectFactory` [http://www.opensymphony.com/xwork/api/com/opensymphony/xwork/ObjectFactory.html] so XWork will look for Spring beans in the root `WebApplicationContext`.

- **ActionAutowiringInterceptor:** use an interceptor to automatically wire an Action's dependencies as they're created.
- **SpringExternalReferenceResolver:** look up Spring beans based on the name defined in an <external-ref> element of an <action> element.

All of these strategies are explained in further detail in WebWork's Documentation [http://wiki.opensymphony.com/display/WW/WebWork+2+Spring+Integration].

Chapter 16. Remoting and web services using Spring

16.1. 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 (TODO)*.

While discussing the remoting capabilities of Spring, we'll use the following domain model and corresponding services:

```
// Account domain object
public class Account implements Serializable{
  private String name;

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

```
// Account service
public interface AccountService {

  public void insertAccount(Account acc);

  public List getAccounts(String name);
}
```

```
// Remote Account service
public interface RemoteAccountService extends Remote {

  public void insertAccount(Account acc) throws RemoteException;

  public List getAccounts(String name) throws RemoteException;
}
```

```
// ... and corresponding implement 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 for Hessian.

16.2. 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.

16.2.1. 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 BeanFactory:

```
<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: We've left out one property, i.e. the `servicePort` property, which is 0 by default. This means an anonymous port will be used to communicate with the service. You can specify a different port if you like.

16.2.2. 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 bean factory, 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.

16.3. Using Hessian or Burlap to remotely call services via HTTP

Hessian offers a binary HTTP-based remoting protocol. It's created by Caucho and more information about Hessian itself can be found at http://www.caucho.com.

16.3.1. 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 an application context 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.

16.3.2. 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`.

16.3.3. Linking in the service on the client

Using the `HessianProxyFactoryBean` 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/AccountService"/>
        <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

That's all there is to it.

16.3.4. 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.

16.3.5. 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.

16.4. 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 [http://jakarta.apache.org/commons/httpclient] for more info.

16.4.1. 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 Http invoker support provides the so-called `org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter`. To expose the `AccountService` (mentioned above), the following configuration needs to be in place:

```
<bean name="/AccountService" class="org.sprfr.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service" ref="accountService"/>
  <property name="serviceInterface" value="example.AccountService"/>
</bean>
```

16.4.2. 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.sprfr.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl" value="http://remotehost:8080/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>
```

16.5. Web Services

Spring has support for:

-

Next to the support listed above, you can also expose your web services using XFire xfire.codehaus.org [http://xfire.codehaus.org]. XFire is a ligthweight SOAP library, currently in development at Codehaus.

16.5.1. 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.

16.5.2. Accessing Web Services

Spring has two factory beans to create web service proxies `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 later 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 magic is done in the spring configuration file as usual:

```
    <bean id="accountWebService" class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
        <property name="serviceInterface">
            <value>example.RemoteAccountService</value>
        </property>
        <property name="wsdlDocumentUrl">
            <value>http://localhost:8080/account/services/accountService?WSDL</value>
        </property>
        <property name="namespaceUri">
            <value>http://localhost:8080/account/services/accountService</value>
        </property>
        <property name="serviceName">
            <value>AccountService</value>
        </property>
        <property name="portName">
            <value>AccountPort</value>
        </property>
    </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 bean="accountWebService"/>
        </property>
    </bean>
```

And 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 e) {
           // ouch
           ...
        }
     }

}
```

We can get rid of the checked RemoteException since Spring supports automatic conversion to its corresponding unchecked `RemoteAccessException`. 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</value>
        </property>
        <property name="portInterface">
            <value>example.RemoteAccountService</value>
        </property>
        ...
    </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(...);
     }

}
```

16.5.3. 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 server-config.wsdd.

We will use Axis to register bean mappings on the client side. To do this we need to subclass Spring Bean factory and register the bean mappings programmatic:

```
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));
        }

}
```

16.5.4. 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 `javax.rpc.xml.handler.Handler` is a callback interface. There is a convenience base class provided in jaxrpc.jar - `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 extend its bean factory and override its `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);
    }

}
```

And 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>
```

16.5.5. Exposing web services using XFire

XFire is a lightweight SOAP library, hosted by Codehaus. At the time of writing (March 2005), XFire is still in development. Although Spring support is stable, lots of features should be added in the future. Exposing XFire is done using an 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 Servlet for that matter). The configuration file is located in the XFire jar and should of course be placed on the classpath of your application archive.

```
<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 you `xfire-servlet.xml`:

```
<beans>
  <bean name="/Echo" class="org.codehaus.xfire.spring.XFireExporter">
    <property name="service" ref="echo">
    <property name="serviceInterface" value="org.codehaus.xfire.spring.Echo"/>
    <property name="serviceBuilder" ref="xfire.serviceBuilder"/>
    <!-- the XFire bean is wired up in the xfire.xml file you've linked in earlier
    <property name="xfire" ref="xfire"/>
  </bean>

  <bean id="echo" class="org.codehaus.xfire.spring.EchoImpl"/>
</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, have a look at the docs.codehaus.org/display/XFIRE/Spring [http://docs.codehaus.org/display/XFIRE/Spring].

16.6. 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.

16.7. 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 problems include the serialization of Hibernate objects in combination with lazily initializing 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 17. Accessing and implementing EJBs

As a lightweight container, Spring is often considered an EJB replacement. We do believe that for many if not most applications and use cases, Spring as a container, combined with its rich supporting functionality in the area of transactions, ORM and JDBC access, is a better choice than implementing equivalent functionality via an EJB container and EJBs.

However, it is important to note that using Spring does not prevent you from using EJBs. In fact, Spring makes it much easier to access EJBs and implement EJBs and functionality within them. Additionally, using Spring to access services provided by EJBs allows the implementation of those services to later transparently be switched between local EJB, remote EJB, or POJO (plain java object) variants, without the client code client code having to be changed.

In this chapter, we look at how Spring can help you access and implement EJBs. Spring provides particular value when accessing stateless session beans (SLSBs), so we'll begin by discussing this.

17.1. Accessing EJBs

17.1.1. Concepts

To invoke a method on a local or remote stateless session bean, client code must normally perform a JNDI lookup to obtain the (local or remote) EJB Home object, then use a 'create' method call on that object to obtain the actual (local or remote) EJB object. One or more methods are then invoked on the EJB.

To avoid repeated low-level code, many EJB applications use the Service Locator and Business Delegate patterns. These are better than spraying JNDI lookups throughout client code, but their usual implementations have significant disadvantages. For example:

- Typically code using EJBs depends on Service Locator or Business Delegate singletons, making it hard to test
- In the case of the Service Locator pattern used without a Business Delegate, application code still ends up having to invoke the create() method on an EJB home, and deal with the resulting exceptions. Thus it remains tied to the EJB API and the complexity of the EJB programming model.
- Implementing the Business Delegate pattern typically results in significant code duplication, where we have to write numerous methods that simply call the same method on the EJB.

The Spring approach is to allow the creation and use of proxy objects, normally configured inside a Spring ApplicationContext or BeanFactory, which act as code-less business delegates. You do not need to write another Service Locator, another JNDI lookup, or duplicate methods in a hand-coded Business Delegate unless you’re adding real value.

17.1.2. Accessing local SLSBs

Assume that we have a web controller that needs to use a local EJB. We’ll follow best practice and use the EJB Business Methods Interface pattern, so that the EJB’s local interface extends a non EJB-specific business methods interface. Let’s call this business methods interface MyComponent.

```
public interface MyComponent {
    ...
```

```
}
```

(One of the main reasons to the Business Methods Interface pattern is to ensure that synchronization between method signatures in local interface and bean implementation class is automatic. Another reason is that it later makes it much easier for us to switch to a POJO (plain java object) implementation of the service if it makes sense to do so) Of course we’ll also need to implement the local home interface and provide a bean implementation class that implements SessionBean and the MyComponent business methods interface. Now the only Java coding we’ll need to do to hook up our web tier controller to the EJB implementation is to expose a setter method of type MyComponent on the controller. This will save the reference as an instance variable in the controller:

```
private MyComponent myComponent;

public void setMyComponent(MyComponent myComponent) {
    this.myComponent = myComponent;
}
```

We can subsequently use this instance variable in any business method in the controller. Now assuming we are obtaining our controller object out of a Spring ApplicationContext or BeanFactory, we can in the same context configure a `LocalStatelessSessionProxyFactoryBean` instance, which will be EJB proxy object. The configuration of the proxy, and setting of the `myComponent` property of the controller is done with a configuration entry such as:

```
<bean id="myComponent"
      class="org.springframework.ejb.access.LocalStatelessSessionProxyFactoryBean">
  <property name="jndiName" value="myComponent"/>
  <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 magic happening behind the scenes, courtesy of the Spring AOP framework, although you aren’t forced to work with AOP concepts to enjoy the results. The `myComponent` bean definition creates a proxy for the EJB, which implements the business method interface. The EJB local home is cached on startup, so there’s only a single JNDI lookup. Each time the EJB is invoked, the proxy invokes the create() method on the local EJB and invokes the corresponding business method on the EJB.

The `myController` bean definition sets the `myController` property of the controller class to this proxy.

This EJB access mechanism delivers huge simplification of application code: The web tier code (or other EJB client code) has no dependence on the use of EJB. If we want to replace this EJB reference with a POJO or a mock object or other test stub, we could simply change the myComponent bean definition without changing a line of Java code. 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 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 XML ApplicationContext variants) you may have a problem if the bean container is loaded before the EJB container loads the target EJB. That is because the JNDI lookup will be performed in the

init method of this class and cached, but the EJB will not have been bound at the target location yet. The solution is to not pre-instantiate this factory object, but allow it to be created on first use. In the XML containers, this is controlled via the `lazy-init` attribute.

Although this will not be of interest to the majority of Spring users, those doing programmatic AOP work with EJBs may want to look at `LocalSlsbInvokerInterceptor`.

17.1.3. Accessing remote SLSBs

Accessing remote EJBs is essentially identical to accessing local EJBs, except that the `SimpleRemoteStatelessSessionProxyFactoryBean` is used. Of course, with or without Spring, remote invocation semantics apply; a call to a method on an object in another VM in another computer does sometimes have to be treated differently in terms of usage scenarios and failure handling.

Spring's EJB client support adds one more advantage over the non-Spring approach. Normally it is problematic for EJB client code to be easily switched back and forth between calling EJBs locally or remotely. This is because the remote interface methods must declare that they throw `RemoteException`, and client code must deal with this, while the local interface methods don't. Client code 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.

17.2. Using Spring 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 Message Driven bean, you 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 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 BeanFactory (or in this case, its ApplicationContext subclass) as part of their lifecycle, which is then available to the EJB (for example, as used in the code above to obtain the POJO service object). The loading is done via a strategy object which is a subclass of `BeanFactoryLocator`. The actual implementation of `BeanFactoryLocator` used by default is `ContextJndiBeanFactoryLocator`, which creates the ApplicationContext from a resource locations specified as a JNDI environment variable (in the case of the EJB classes, at `java:comp/env/ejb/BeanFactoryPath`). If there is a need to change the BeanFactory/ApplicationContext loading strategy, the default `BeanFactoryLocator` implementation used may be 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 BeanFactory/ApplicationContext instance (which is the normal case) will have to manually call `unloadBeanFactory()` and `loadBeanFactory` from `ejbPassivate` and `ejbActivate`, respectively, to unload and reload the BeanFactory on passivation and activation, since it can not be saved by the EJB container.

The default usage of `ContextJndiBeanFactoryLocator` to load an ApplicationContext for the use of the EJB is adequate for some situations. However, it is problematic when the ApplicationContext is loading a number of beans, or the 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 `ContextSingletonBeanFactoryLocatore`, which can load and use a shared BeanFactory or ApplicationContext to be used by multiple EJBs or other clients. Doing this is relatively simple, by adding code similar to this to the EJB:

```
   /**
    * Override default BeanFactoryLocator implementation
    *
    * @see javax.ejb.SessionBean#setSessionContext(javax.ejb.SessionContext)
    */
   public void setSessionContext(SessionContext sessionContext) {
       super.setSessionContext(sessionContext);
       setBeanFactoryLocator(ContextSingletonBeanFactoryLocator.getInstance());
       setBeanFactoryLocatorKey(ServicesConstants.PRIMARY_CONTEXT_ID);
   }
```

Please see the respective JavaDocs for `BeanFactoryLocator` and `ContextSingletonBeanFactoryLocatore` for more information on their usage.

Chapter 18. JMS

18.1. 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, production and consumption of messages. In a J2EE environment, the ability to consume messages asynchronously is provided for by message-driven beans while in a standalone application this is provided for by the creation of MessageListeners or ConnectionConsumers. The functionality in JmsTemplate is focused on producing messages. Future releases of Spring will address asynchronous message consumption in a standalone environment.

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 principal 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. This allows for integration of JMS as a transactional resource into Spring's transaction management mechanisms.

18.2. 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. Consequentially, a client application was domain specific in the 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`.

The JMS 1.1 specification was released in April 2002 and incorporated as part of J2EE 1.4 in November 2003. As a result, most application servers that are currently in use are only required to support JMS 1.0.2.

18.3. JmsTemplate

Two implementations of the JmsTemplate are provided. The class `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 QOS different default values than those specified in the JMS specification. Therefore, in order to provide consistent management of QOS values, the JmsTemplate must be specifically enabled to use its own QOS values by setting the boolean property `isExplicitQosEnabled` to true.

18.3.1. ConnectionFactory

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 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 comes from JNDI.

18.3.2. 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 7`. The `JmsTransactionManager` binds a Connection/Session pair from the specified ConnectionFactory to the thread. However, in a J2EE environment the ConnectionFactory will pool connections and sessions, so the instances that are bound to the thread depend on the pooling behavior. In a standalone environment, using Spring's `SingleConnectionFactory` will result in a using a single JMS Connection and each transaction having its own Session. The `JmsTemplate` can also be used with the `JtaTransactionManager` and an XA-capable JMS ConnectionFactory for performing distributed transactions.

Reusing code across a managed and unmanaged transactional environment can be confusing when using JMS

API to create a `Session` from a Connection. This is because the JMS API only 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 in the responsibility of the environments 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 though the use of the properties `SessionTransacted` and `SessionAcknowledgeMode`. When using a `PlatformTransactionManager` with `JmsTemplate`, the template will always be given a transactional JMS Session.

18.3.3. Destination Management

Destinations, like ConnectionFactories, are JMS administered objects that can be stored and retrieved in JNDI. When configuring a Spring application context one 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 can not 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 resolving dynamic destination via implementations of `DestinationResolver`.

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.

18.4. Using the JmsTemplate

To get started using the JmsTemplate you need to select either the JMS 1.0.2 implementation `JmsTemplate102` or the JMS 1.1 implementation `JmsTemplate`. Check your JMS provider to determine what version is supported.

18.4.1. 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) {
    jt = new JmsTemplate102(cf, false);
  }

  public void setQueue(Queue q) {
    queue = q;
  }

  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.

18.4.2. Synchronous Receiving

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.

18.4.3. 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 your application code can focus on the business object that is being sent or received via JMS and not bother 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 bust an existing XML marshalling packages, 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 interface `MessagePostProcessor` gives you access to the message after it has been converted, but before it is sent. The example below shows 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(35));
  jt.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:35}
  }
}
```

18.4.4. 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.

Chapter 19. JMX Support

19.1. Introduction

The JMX support in Spring provides you with the features to easily and transparently integrate your Spring application into a JMX infrastructure. Specifically, Spring JMX provides 4 core features:

- Automatic Registration of any Spring bean as a JMX MBean
- Flexible mechanism for controlling the management interface of your beans
- Declarative exposure of MBeans over remote, JSR-160 connectors
- 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.

19.2. Exporting your Beans to JMX

The core class in the Spring JMX framework is the `MBeanExporter`. This class is responsible for taking your Spring beans and registering them with the JMX `MBeanServer`. For example, consider the simple bean class shown below:

```
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 a JMX MBean you simply configure an instance of the `MBeanExporter` class in your configuration file and pass in the bean as shown below:

```
<beans>

  <bean id="exporter" class="org.springframework.jmx.export.MBeanExporter">
    <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>
```

Here, the important definition is the `exporter` bean. The `beans` property is used to tell the `MBeanExporter` which of your beans should be exported to the JMX `MBeanServer`. The `beans` property is of type `Map`, and thus you use the `<map>` and `<entry>` tags to configure the beans to be exported. In the default configuration, the key of an entry in of the `Map` is used as the `ObjectName` for the bean that is the value of that entry. This behaviour can be changed as described in section Section 19.4, “Controlling the ObjectNames for your Beans”.

With this configuration the `testBean` bean is exposed as a JMX MBean under the `ObjectName` `bean:name=testBean1`. All public properties of the bean are exposed as attributes and all public methods (expect those defined in `Object`) are exposed as operations.

19.2.1. Creating an MBeanServer

The configuration shown above assumes that the application is running in an environment that has one and only one `MBeanServer` already running. In this case, Spring will locate the running `MBeanServer` and register your beans with that. This is useful when your application is running inside a container such as Tomcat or IBM WebSphere that has its own `MBeanServer`.

However, this is approach is no use for standalone environment, or when running inside a container that does not provide an `MBeanServer`. To overcome this you can create an `MBeanServer` instance declaratively by adding an instance of `org.springframework.jmx.support.MBeanServerFactoryBean` to your configuration. You can also ensure that this `MBeanServer` is used by using `MBeanServerFactoryBean` to set the server property of the `MBeanExporter`. This is shown below:

```
<beans>

  <bean id="mbeanServer" class="org.springframework.jmx.support.MBeanServerFactoryBean"/>

  <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`, `MBeanExporter` will not attempt to locate a running `MBeanServer`. For this to work correctly, you must have a JMX implementation on your classpath.

19.2.2. 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 `BeanFactory` until the first invocation on the proxy occurs.

19.2.3. 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 section Section 19.4, “Controlling the ObjectNames for your Beans”.

19.3. Controlling the Management Interface of Your Beans

In the previous example, you had little control over the management interface of your bean with all the public properties and methods being exposed. To solve this problem, Spring JMX provides a comprehensive and extensible mechanism for controlling the management interfaces of your beans.

19.3.1. 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 an interface that exposes all public properties and methods as you saw in the previous example. Spring provides two additional implementations of the `MBeanInfoAssembler` interface that allow you to control the management interface using source level metadata or any arbitrary interface.

19.3.2. 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 of `JmxAttributeSource` for it to function correctly. For this 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 should be marked with a `ManagedOperation` attribute and each property you wish to expose should be marked with a `ManagedAttribute` attribute. When marking properties you can omit either 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 `ManagedResoure` 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 Section 19.3.4, “Source-Level Metadata Types”.

You will also notice that both the age and name properties are marked 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, and for the `age` attribute to 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 casue 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">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="assembler">
      <ref local="assembler"/>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </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 local="attributeSource"/>
    </property>
  </bean>

</beans>
```

Here you can see that a `MetadataMBeanInfoAssembler` bean has been configured with an instance of `AttributesJmxAttributeSource` 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.

19.3.3. 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

`JmxAttributeSource`, `AnnotationsJmxAttributeSource`, that allows the `MBeanInfoAssembler` to read them.

The example below shows a bean with a JDK 5.0 annotation defined management interface:

```
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 Together")
  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 benefit of compile-time checking.

19.3.4. Source-Level Metadata Types

The following source level metadata types are available for use in Spring JMX:

Table 19.1. Source-Level Metadata Types

Purpose	Commons Attributes Attribute	JDK 5.0 Annotation	Attribute / Annotation Type
Mark all instances of a Class as JMX managed	`ManagedResource`	`@ManagedResource`	Class

Purpose	Commons Attributes Attribute	JDK 5.0 Annotation	Attribute / Annotation Type
resources			
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`	`@ManagedOperationParameter` and `@ManagedOperationParameters`	Method

The following configuration parameters are available for use on these source-level metadata types:

Table 19.2. 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` descriptor field	`ManagedResource`
`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	`ManagedOperationParameter`

Parameter	Description	Applies to
	operation parameter	
`index`	Sets the index of an operation parameter	`ManagedOperationParameter`

19.3.5. 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 `AutodetectCapableMBeanInfo` 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">
    <property name="assembler" ref="assembler"/>
          <property name="autodetect" value="true"/>
  </bean>

  <bean id="bean:name=testBean1" 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"/>

  <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 solve this problem by changing the default behavior for `ObjectName` creation as defined in section Section 19.4, “Controlling the ObjectNames for your Beans”.

19.3.6. 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">
          <ref local="testBean"/>
        </entry>
      </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</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </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` via the `interfaceMappings` property, where the key of each entry is the bean name and the value of each entry is a comma-seperated list of interface names to use for that bean.

If no management interface is specified through either the `managedInterfaces` or `interfaceMappings` properties, then `InterfaceBasedMBeanInfoAssembler` will reflect on the bean and use all interfaces implemented by that bean to create the management interface.

19.3.7. 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">
          <ref local="testBean"/>
        </entry>
      </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` 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.

19.4. Controlling the `ObjectName`s for your Beans

Behind the scenes, the `MBeanExporter` delegates to an implementation of the `ObjectNamingStrategy` to obtain `ObjectName`s 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: `IdentityNamingStrategy` that builds an `ObjectName` based on the identity of the bean and `MetadataNamingStrategy` that uses the source level metadata to obtain the `ObjectName`.

19.4.1. Reading `ObjectName`s from `Properties`

You can configure your own `KeyNamingStrategy` instance and configure it to read `ObjectName`s 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 is used as the `ObjectName`.

19.4.2. 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>
```

19.5. Exporting your Beans with JSR-160 Connectors

For remote access, Spring JMX module offers two `FactoryBean` implementations inside the `org.springframework.jmx.support` package for creating server-side and client-side connectors.

19.5.1. 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: 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 the 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>
```

For more information on these properties consult the JavaDoc. For information of meaning of the environment variables, consult the JavaDoc for

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>
```

19.5.2. 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>
```

19.5.3. 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(IIOP and JRMP) and the optional JMXMP. By using other providers or implementations like MX4J [http://mx4j.sourceforge.net] you can take advantage of protocols like SOAP, Hessian, Burlap over simple HTTP or SSL and other:

```
<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>
```

For this example, MX4J 3.0.0 was used. See the official MX4J documentation for more information.

19.6. 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 to configure a proxy for an MBean running in the local `MBeanServer`:

```
<bean id="proxy" class="org.springframework.jmx.access.MBeanProxyFactoryBean">
    <property name="objectName">
      <value>bean:name=testBean</value>
    </property>
    <property name="proxyInterface">
      <value>org.springframework.jmx.IJmxTestBean</value>
    </property>
</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` allowing 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"/>
</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 pass on all invocations to the `MBeanServer` via this `MBeanServerConnection`.

Chapter 20. JCA CCI

20.1. Introduction

J2EE provides a specification to standardize access to EIS: 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 thid 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's Spring general resource and transaction management facilities.

Important 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.

20.2. Configuring CCI

20.2.1. 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 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).

Important 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.

20.2.2. 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 it 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</value>
  </property>
</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</value></property>
  <property name="connectionURL"><value>tcp://localhost/</value></property>
  <property name="portNumber"><value>2006</value></property>
</bean>

<bean id="eciConnectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory">
    <ref local="eciManagedConnectionFactory"/>
  </property>
</bean>
```

Important 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.

20.2.3. 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 provided 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</value>
  </property>
  <property name="driverName"><value>org.hsqldb.jdbcDriver</value></property>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory">
    <ref local="managedConnectionFactory"/>
  </property>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory">
    <ref bean="targetConnectionFactory"/>
  </property>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user"><value>sa</value></property>
      <property name="password"><value/></property>
    </bean>
  </property>
</bean>
```

20.2.4. 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 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</value></property>
  <property name="connectionURL"><value>tcp://localhost/</value></property>
  <property name="portNumber"><value>2006</value></property>
</bean>

<bean id="targetEciConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory">
    <ref local="eciManagedConnectionFactory"/>
  </property>
</bean>

<bean id="eciConnectionFactory"
    class="org.springframework.jca.cci.connection.SingleConnectionFactory">
  <property name="targetConnectionFactory">
    <ref local="targetEciConnectionFactory"/>
  </property>
</bean>
```

Important 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`.

20.3. Using Spring's CCI access support

20.3.1. 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.

```
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);
  }
};
```

20.3.2. CciTemplate

This 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 { ... }
```

```
  ...
}
```

20.3.3. 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() { ... }
  ...
}
```

20.3.4. 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 a previous section. The `outputRecordCreator` property must be directly specified on the `CciTemplate`. This could be done in the application code:

```
cciTemplate.setOutputRecordCreator(new EciOutputRecordCreator());
```

or 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 local="eciConnectionFactory"/>
  </property>
  <property name="outputRecordCreator">
    <ref local="eciOutputRecordCreator"/>
  </property>
</bean>
```

`Note`: As the `CciTemplate` class is thread-safe, it will usually be configured as a shared instance.

20.3.5. Summary

The following table summarizes the mechanism of the `CciTemplate` class and the corresponding methods called on the CCI `Interaction` interface:

Table 20.1. 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)

20.3.6. 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 and newly created inside every callback method. This is completely up to the DAO implementation.

20.3.7. 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 {
            ...
          }
        });
    }
    return output;
  }
}
```

Important 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</value></property>
  <property name="connectionURL"><value>local:</value></property>
  <property name="userName"><value>CICSUSER</value></property>
  <property name="password"><value>CICS</value></property>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory">
    <ref local="managedConnectionFactory"/>
  </property>
</bean>

<bean id="component" class="mypackage.MyDaoImpl">
  <property name="connectionFactory"><ref local="connectionFactory"/></property>
</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</value></property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory"><ref local="connectionFactory"/></property>
</bean>
```

20.4. 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.

20.4.1. 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);
...
```

20.4.2. 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;
  ...
}
```

20.4.3. 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 previous "automatic output record generation" section.

20.4.4. Summary

The operation object approach uses records in the same manner as the `CciTemplate` class.

Table 20.2. Usage of Interaction execute methods

MappingRecordOperation method signature	MappingRecordOperarion 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)

20.4.5. 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</value>
  </property>
  <property name="driverName"><value>org.hsqldb.jdbcDriver</value></property>
</bean>

<bean id="targetConnectionFactory"
    class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory">
    <ref local="managedConnectionFactory"/>
  </property>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory">
    <ref bean="targetConnectionFactory"/>
  </property>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user"><value>sa</value></property>
      <property name="password"><value/></property>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory"><ref local="connectionFactory"/></property>
</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</value></property>
</bean>

<bean id="connectionFactory"
    class="org.springframework.jca.cci.connection.ConnectionSpecConnectionFactoryAdapter">
  <property name="targetConnectionFactory">
    <ref bean="targetConnectionFactory"/>
  </property>
  <property name="connectionSpec">
    <bean class="com.sun.connector.cciblackbox.CciConnectionSpec">
      <property name="user"><value>sa</value></property>
      <property name="password"><value/></property>
    </bean>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory"><ref local="connectionFactory"/></property>
</bean>
```

20.4.6. 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</value></property>
  <property name="connectionURL"><value>local:</value></property>
  <property name="userName"><value>CICSUSER</value></property>
  <property name="password"><value>CICS</value></property>
</bean>

<bean id="connectionFactory" class="org.springframework.jca.support.LocalConnectionFactoryBean">
  <property name="managedConnectionFactory">
    <ref local="managedConnectionFactory"/>
  </property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory"><ref local="connectionFactory"/></property>
</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</value></property>
</bean>

<bean id="component" class="MyDaoImpl">
  <property name="connectionFactory"><ref local="connectionFactory"/></property>
</bean>
```

20.5. Transactions

JCA specifies several levels of transaction suppot 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 CICS ECI connector), global transactions (for example with 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</value>
  </property>
</bean>

<bean id="eciTransactionManager"
    class="org.springframework.jca.cci.connection.CciLocalTransactionManager">
  <property name="connectionFactory">
    <ref local="eciConnectionFactory" />
  </property>
</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 transaction management chapter.

Chapter 21. Sending Email with Spring mail abstraction layer

21.1. Introduction

Spring provides a higher level of abstraction for sending electronic mail which shields the user from the specifics of underlying mailing system and is responsible for a low level resource handling on behalf of the client.

21.2. Spring mail abstraction structure

The main package of Spring mail abstraction layer is `org.springframework.mail` package. It contains central interface for sending emails called `MailSender` and the *value object* which encapsulates properties of a simple mail such as *from*, *to*, *cc*, *subject*, *text* called `SimpleMailMessage`. This package also 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 JavaDocs for more information on mail exception hierarchy.

Spring also provides a sub-interface of `MailSender` for specialized *JavaMail* features such as MIME messages, namely `org.springframework.mail.javamail.JavaMailSender` It also provides a callback interface for preparation of JavaMail MIME messages, namely `org.springframework.mail.javamail.MimeMessagePreparator`

MailSender:

```
public interface MailSender {

    /**
     * Send the given simple mail message.
     * @param simpleMessage message to send
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(SimpleMailMessage simpleMessage) throws MailException;

    /**
     * Send the given array of simple mail messages in batch.
     * @param simpleMessages messages to send
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(SimpleMailMessage[] simpleMessages) throws MailException;

}
```

JavaMailSender:

```
public interface JavaMailSender extends MailSender {

    /**
     * Create a new JavaMail MimeMessage for the underlying JavaMail Session
     * of this sender. Needs to be called to create MimeMessage instances
     * that can be prepared by the client and passed to send(MimeMessage).
     * @return the new MimeMessage instance
     * @see #send(MimeMessage)
     * @see #send(MimeMessage[])
     */
    public MimeMessage createMimeMessage();

    /**
     * Send the given JavaMail MIME message.
     * The message needs to have been created with createMimeMessage.
```

```
     * @param mimeMessage message to send
     * @throws MailException in case of message, authentication, or send errors
     * @see #createMimeMessage
     */
    public void send(MimeMessage mimeMessage) throws MailException;

    /**
     * Send the given array of JavaMail MIME messages in batch.
     * The messages need to have been created with createMimeMessage.
     * @param mimeMessages messages to send
     * @throws MailException in case of message, authentication, or send errors
     * @see #createMimeMessage
     */
    public void send(MimeMessage[] mimeMessages) throws MailException;

    /**
     * Send the JavaMail MIME message prepared by the given MimeMessagePreparator.
     * Alternative way to prepare MimeMessage instances, instead of createMimeMessage
     * and send(MimeMessage) calls. Takes care of proper exception conversion.
     * @param mimeMessagePreparator the preparator to use
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(MimeMessagePreparator mimeMessagePreparator) throws MailException;

    /**
     * Send the JavaMail MIME messages prepared by the given MimeMessagePreparators.
     * Alternative way to prepare MimeMessage instances, instead of createMimeMessage
     * and send(MimeMessage[]) calls. Takes care of proper exception conversion.
     * @param mimeMessagePreparators the preparator to use
     * @throws MailException in case of message, authentication, or send errors
     */
    public void send(MimeMessagePreparator[] mimeMessagePreparators) throws MailException;

}
```

MimeMessagePreparator:

```
public interface MimeMessagePreparator {

    /**
     * Prepare the given new MimeMessage instance.
     * @param mimeMessage the message to prepare
     * @throws MessagingException passing any exceptions thrown by MimeMessage
     * methods through for automatic conversion to the MailException hierarchy
     */
    void prepare(MimeMessage mimeMessage) throws MessagingException;

}
```

21.3. Using Spring mail abstraction

Let's assume there is a business interface called `OrderManager`

```
public interface OrderManager {

    void placeOrder(Order order);
}
```

and there is a use case that says that an email message with order number would need to be generated and sent to a customer placing that order. So for this purpose we want to use `MailSender` and `SimpleMailMessage`

Please note that as usual, we work with interfaces in the business code and let Spring IoC container take care of wiring of all the collaborators for us.

Here is the implementation of `OrderManager`

```
import org.springframework.mail.MailException;
import org.springframework.mail.MailSender;
import org.springframework.mail.SimpleMailMessage;
```

```
public class OrderManagerImpl implements OrderManager {

    private MailSender mailSender;
    private SimpleMailMessage message;

    public void setMailSender(MailSender mailSender) {
        this.mailSender = mailSender;
    }

    public void setMessage(SimpleMailMessage message) {
        this.message = message;
    }

    public void placeOrder(Order order) {

        //... * Do the business calculations....
        //... * Call the collaborators to persist the order

        //Create a thread safe "sandbox" of the message
        SimpleMailMessage msg = new SimpleMailMessage(this.message);
        msg.setTo(order.getCustomer().getEmailAddress());
        msg.setText(
            "Dear "
                + order.getCustomer().getFirstName()
                + order.getCustomer().getLastName()
                + ", thank you for placing order. Your order number is "
                + order.getOrderNumber());
        try{
            mailSender.send(msg);
        }
        catch(MailException ex) {
            //log it and go on
            System.err.println(ex.getMessage());
        }
    }
}
```

Here is what the bean definitions for the code above would look like:

```
<bean id="mailSender" class="org.springframework.mail.javamail.JavaMailSenderImpl">
  <property name="host"><value>mail.mycompany.com</value></property>
</bean>

<bean id="mailMessage" class="org.springframework.mail.SimpleMailMessage">
  <property name="from"><value>customerservice@mycompany.com</value></property>
  <property name="subject"><value>Your order</value></property>
</bean>

<bean id="orderManager" class="com.mycompany.businessapp.support.OrderManagerImpl">
  <property name="mailSender"><ref bean="mailSender"/></property>
  <property name="message"><ref bean="mailMessage"/></property>
</bean>
```

Here is the implementation of `OrderManager` using `MimeMessagePreparator` callback interface. Please note that the mailSender property is of type `JavaMailSender` in this case in order to be able to use JavaMail MimeMessage:

```
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;

import javax.mail.internet.MimeMessage;
import org.springframework.mail.MailException;
import org.springframework.mail.javamail.JavaMailSender;
import org.springframework.mail.javamail.MimeMessagePreparator;

public class OrderManagerImpl implements OrderManager {

    private JavaMailSender mailSender;

    public void setMailSender(JavaMailSender mailSender) {
        this.mailSender = mailSender;
```

```
    }

    public void placeOrder(final Order order) {

        //... * Do the business calculations....
        //... * Call the collaborators to persist the order

        MimeMessagePreparator preparator = new MimeMessagePreparator() {
            public void prepare(MimeMessage mimeMessage) throws MessagingException {
                mimeMessage.setRecipient(Message.RecipientType.TO,
                        new InternetAddress(order.getCustomer().getEmailAddress()));
                mimeMessage.setFrom(new InternetAddress("mail@mycompany.com"));
                mimeMessage.setText(
                    "Dear "
                        + order.getCustomer().getFirstName()
                        + order.getCustomer().getLastName()
                        + ", thank you for placing order. Your order number is "
                        + order.getOrderNumber());
            }
        };
        try{
            mailSender.send(preparator);
        }
        catch (MailException ex) {
            //log it and go on
            System.err.println(ex.getMessage());
        }
    }
}
```

If you want to use JavaMail MimeMessage to the full power, the `MimeMessagePreparator` is available at your fingertips.

Please note that the mail code is a crosscutting concern and is a perfect candidate for refactoring into a custom Spring AOP advice, which then could easily be applied to OrderManager target. Please see the AOP chapter.

21.3.1. Pluggable MailSender implementations

Spring comes with two MailSender implementations out of the box - the JavaMail implementation and the implementation on top of Jason Hunter's *MailMessage* class that's included in http://servlets.com/cos (com.oreilly.servlet). Please refer to JavaDocs for more information.

21.4. Using the JavaMail MimeMessageHelper

One of the components that comes in pretty handy when dealing with JavaMail messages is the `org.springframework.mail.javamail.MimeMessageHelper`. It prevents you from having to use the nasty APIs the the `javax.mail.internet` classes. A couple of possible scenarios:

21.4.1. Creating a simple MimeMessage and sending it

Using the MimeMessageHelper it's pretty easy to setup and send a MimeMessage:

```
// of course you would setup the mail sender using
// DI in any real-world cases
JavaMailSenderImpl sender = new JavaMailSenderImpl();
sender.setHost("mail.host.com");

MimeMessage message = sender.createMimeMesage();
MimeMessageHelper helper = new MimeMessageHelper(message);
helper.setTo("test@host.com");
helper.setText("Thank you for ordering!");
```

```
sender.send(message);
```

21.4.2. Sending attachments and inline resources

Email allow for attachments, but also for inline resources in multipart messages. Inline resources could for example be images or stylesheet you want to use in your message, but don't want displayed as attachment. The following 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.createMimeMesage();

// 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);

// if you would need to include the file as an attachment, use
// addAttachment() methods on the MimeMessageHelper

sender.send(message);
```

Inline resources are added to the mime message using the Content-ID specified as you've seen just now (`identifier1234` in this case). The order in which you're adding the text and the resource are VERY important. First add the text and after that the resources. If you're doing it the other way around, it won't work!

Chapter 22. Scheduling jobs using Quartz or Timer

22.1. Introduction

Spring features integration classes for scheduling support. Currently, Spring supports the Timer, part of the JDK since 1.3, and the Quartz Scheduler (http://www.quartzscheduler.org). Both schedulers are set up using a FactoryBean with optional references to Timers or Triggers, 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 normal `MethodInvokingFactoryBeans`).

22.2. Using the OpenSymphony Quartz Scheduler

Quartz uses `Triggers`, `Jobs` and `JobDetail` ro 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 usage of Quartz within Spring-based applications.

22.2.1. Using the JobDetailBean

`JobDetail` objects contain all information needed to run a job. Spring provides a so-called `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 as 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 equals the bean name of the job detail bean (in the example above, this is `exampleJob`).

22.2.2. 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` being called on the exampleBusinessObject (see below):

```
public class BusinessObject {

  // 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 objects 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.

22.2.3. Wiring up jobs using triggers and the SchedulerFactoryBean

We've created job details, jobs and we've 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 exposing properties to set the triggers.

SchedulerFactoryBean schedules the actual jobs with those triggers.

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>
```

OK, 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 JavaDoc (http://www.springframework.org/docs/api/org/springframework/scheduling/quartz/SchedulerFactoryBean.html) for more information.

22.3. Using JDK Timer support

The other way to schedule jobs in Spring is using JDK Timer objects. More information about Timers themselves can be found at http://java.sun.com/docs/books/tutorial/essential/threads/timer.html. The concepts discussed above also apply to the Timer support. You can create custom timers or use the timer that invokes methods. Wiring timers has to be done using the TimerFactoryBean.

22.3.1. 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>
```

Letting the task only run once can be done by changing the `period` property to -1 (or some other negative value)

22.3.2. 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` being called on the `exampleBusinessObject` (see below):

```
public class BusinessObject {

  // properties and collaborators

  public void doIt() {
    // do the actual work
  }
}
```

Changing the reference of the above example (in which the ScheduledTimerTask is mentioned) to the `doIt` will result in this task being executed.

22.3.3. 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>
```

That's all!

Chapter 23. Testing

23.1. Unit testing

You don't need this manual to help you write effective unit tests for Spring-based applications.

One of the main benefits of Dependency Injection is that your code should 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 or many other valuable testing techniques, to test your code in isolation. If you follow the architecture recommendations around Spring--for example, those in *J2EE without EJB*--you will find that the resulting clean layering will also greatly facilitate 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 will run extremely quickly, as there is no runtime infrastructure to set up, whether application server, database, ORM tool etc. Thus emphasizing true unit tests will boost your productivity.

23.2. Integration testing

However, it's also important to be able to perform some integration testing without deployment to your application server. This will test things such as:

- Correct wiring of your Spring contexts.
- Data access using JDBC or ORM tool--correctness of SQL statements. For example, you can test your DAO implementation classes.

Thus Spring provides valuable support for integration testing, in the `spring-mock.jar`. This can be thought of as a significantly superior alternative to in-container testing using tools such as Cactus.

The `org.springframework.test` package provides valuable superclasses for integration tests using a Spring container, but not dependent on an application server or other deployed environment. Such tests can run in JUnit--even in an IDE--without any special deployment step. They will be slower to run than unit tests, but much faster to run than Cactus tests or remote tests relying on deployment to an application server.

The superclasses in this package provide the following functionality:

- Context caching.
- Dependency Injection for test classes.
- Transaction management appropriate to tests.
- Inherited instance variables useful for testing.

Numerous Interface21 and other projects since late 2004 have demonstrated the power and utility of this approach. Let's look at some of the important areas of functionality in detail.

23.2.1. Context management and caching

The `org.springframework.test` package provides support for consistent loading of Spring contexts, and caching of loaded contexts. The latter 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 them, and incurring that cost before running every test case will greatly reduce productivity.

Thus, `AbstractDependencyInjectionSpringContextTests` has an abstract protected method that subclasses must implement, to provide the location of contexts:

```
protected abstract String[] getConfigLocations();
```

This should provide a list of the context locations--typically on the classpath--used to configure the application. This will be the same, or nearly the same, as the list of config locations specified in web.xml or other deployment configuration.

By default, once loaded, the set of configs will be reused for each test case. Thus the setup cost will be incurred only once, 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 it to reload the configurations and rebuild the application context before executing the next test case.

23.2.2. Dependency Injection of test class instances

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.

The superclasses use *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 `getBean()`.

If you don't want Setter Injection applied to your test cases, don't declare any setters. Or extend `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.

23.2.3. Transaction management

One common problem in tests that access a real database is their effect 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 persistence data--can't 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 case. 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 a `DataSource` bean definition--again, with any name--is 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.)

If you want a transaction to commit--unusual, but useful if you want a particular test to populate the database, for example--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 behaviour 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.

Note that these test support classes are designed to work with a single database.

23.2.4. Convenience variables

When you extend org.springframework.test package you will have access to the following protected instance variables:

- `applicationContext` (`ConfigurableApplicationContext`): inherited from AbstractDependencyInjectionSpringContextTests. 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.

23.2.5. Example

The PetClinic sample application included with the Spring distribution illustrates the use of these test superclasses (Spring 1.1.5 and above).

Most test functionality is included in `AbstractClinicTests`, for which a partial listing is shown belong:

```
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 `org.springframework.AbstractTransactionalDataSourceSpringContextTests`, from which it inherits Dependency Injection and transactional behaviour.
- 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 behaviour 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 three data access technologies--JDBC, Hibernate and Apache OJB. Thus `AbstractClinicTests` does not specify the context locations--this is deferred to subclasses, that implement the necessary protected abstract method from `AbstractDependencyInjectionSpringContextTests`.

For example, the JDBC implementation of the PetClinic tests contains the following method:

```
public 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 behaviour 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.

23.2.6. Running integration tests

Integration tests naturally have more environmental dependencies than plain unit tests. Such integration testing is an additional form of testing, not a substitute for unit testing.

The main dependency will typically be on a development database containing a complete schema used by the application. This may also contain test data, set up by a a tool such as a DBUnit, or an import using your database's tool set.

Appendix A. `spring-beans.dtd`

```
<?xml version="1.0" encoding="UTF-8"?>

<!--
        Spring XML Beans DTD
        Authors: Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu

        This defines a simple and consistent way of creating a namespace
        of JavaBeans objects, managed by a Spring BeanFactory, read by
        XmlBeanDefinitionReader (with DefaultXmlBeanDefinitionParser).

        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.

        Bean instances can be "singletons" (shared instances) or "prototypes"
        (independent instances). Further scopes are supposed to be built on top
        of the core BeanFactory infrastructure and are therefore not part of it.

        References among beans are supported, i.e. 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.

        As the format is simple, a DTD is sufficient, and there's no need
        for a schema at this point.

        XML documents that conform to this DTD should declare the following doctype:

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


<!--
        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-dependency-check (none | objects | simple | all) "none">
<!ATTLIST beans default-autowire (no | byName | byType | constructor | autodetect) "no">

<!--
        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>


<!--
	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?,
	(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 or commas.
-->
<!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, singleton, lazy init.
-->
<!ATTLIST bean parent 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.
-->
<!ATTLIST bean abstract (true | false) "false">

<!--
        Is this bean a "singleton" (one shared instance, which will
        be returned by all calls to getBean() with the id),
        or a "prototype" (independent instance resulting from each call to
        getBean(). Default is singleton.

        Singletons are most commonly used, and are ideal for multi-threaded
        service objects.
-->
<!ATTLIST bean singleton (true | false) "true">

<!--
        If this bean should be lazily initialized.
        If false, it will get instantiated on startup by bean factories
        that perform eager initialization of singletons.
-->
<!ATTLIST bean lazy-init (true | false | default) "default">

<!--
        Optional attribute controlling whether to "autowire" bean properties.
        This is an automagical process in which bean references don't need to be coded
        explicitly in the XML bean definition file, but Spring works out dependencies.

        There are 5 modes:

        1. "no"
        The traditional Spring default. No automagical wiring. Bean references
        must be defined in the XML file via the <ref> element. We recommend this
        in most cases as it makes documentation more explicit.

        2. "byName"
        Autowiring by property name. If a bean of class Cat exposes a dog property,
        Spring will try to set this to the value of the bean "dog" in the current factory.
        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 behaviour can be combined with dependency
        checking, which will be performed after all autowiring has been completed.
-->
<!ATTLIST bean autowire (no | byName | byType | constructor | autodetect | default) "default">

<!--
        Optional attribute controlling whether to check whether all this
        beans dependencies, expressed in its properties, are satisfied.
        Default is no dependency checking.

        "simple" type dependency checking includes primitives and String
        "object" includes collaborators (other beans in the factory)
        "all" includes both types of dependency checking
-->
<!ATTLIST bean dependency-check (none | objects | simple | all | default) "default">
```

```
<!--
	The names of the beans that this bean depends on being initialized.
	The bean factory will guarantee that these beans get initialized before.

	Note that dependencies are normally expressed through bean properties or
	constructor arguments. This property should just be necessary for other kinds
	of dependencies like statics (*ugh*) or database preparation on startup.
-->
<!ATTLIST bean depends-on CDATA #IMPLIED>

<!--
	Optional attribute for the name of the custom initialization method
	to invoke after setting bean properties. The method must have no arguments,
	but may throw any exception.
-->
<!ATTLIST bean init-method CDATA #IMPLIED>

<!--
	Optional attribute for the name of the custom destroy method to invoke
	on bean factory shutdown. The method must have no arguments,
	but may throw any exception. Note: Only invoked on singleton beans!
-->
<!ATTLIST bean destroy-method CDATA #IMPLIED>

<!--
	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?,
        (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 helper tools.
-->
<!ELEMENT idref EMPTY>
```

```
<!--
	ID refs must specify a name of the target bean.
	The "bean" attribute can reference any name from any bean in the context,
	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 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)*
)>

<!--
	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)*
)>


<!--
	A Spring map is a mapping from a string key to object.
	Maps may be empty.
-->
<!ELEMENT map (
	(entry)*
)>

<!--
	A map entry can be an inner bean, ref, 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)*
)>

<!--
        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>
```