

17.6. Tapestry 3.x and 4.x	504
Injecting Spring-managed beans	505
Dependency Injecting Spring Beans into Tapestry pages	506
Component definition files	507
Adding abstract accessors	508
Dependency Injecting Spring Beans into Tapestry pages - Tapestry 4.x style	510
17.7. Further Resources	511
18. Portlet MVC Framework	512
18.1. Introduction	512
Controllers - The C in MVC	513
Views - The V in MVC	513
Web-scoped beans	514
18.2. The DispatcherPortlet	514
18.3. The ViewRendererServlet	516
18.4. Controllers	517
AbstractController and PortletContentGenerator	517
Other simple controllers	519
Command Controllers	519
PortletWrappingController	520
18.5. Handler mappings	520
PortletModeHandlerMapping	521
ParameterHandlerMapping	522
PortletModeParameterHandlerMapping	522
Adding HandlerInterceptors	523
HandlerInterceptorAdapter	523
ParameterMappingInterceptor	523
18.6. Views and resolving them	524
18.7. Multipart (file upload) support	524
Using the PortletMultipartResolver	525
Handling a file upload in a form	525
18.8. Handling exceptions	528
18.9. Annotation-based controller configuration	529
Setting up the dispatcher for annotation support	529
Defining a controller with <code>@Controller</code>	529
Mapping requests with <code>@RequestMapping</code>	530
Supported handler method arguments	531
Binding request parameters to method parameters with <code>@RequestParam</code>	533
Providing a link to data from the model with <code>@ModelAttribute</code>	534
Specifying attributes to store in a Session with <code>@SessionAttributes</code>	534
Customizing WebDataBinder initialization	535
Customizing data binding with <code>@InitBinder</code>	535
Configuring a custom WebBindingInitializer	535
18.10. Portlet application deployment	536
VI. Integration	537

19. Remoting and web services using Spring	538
19.1. Introduction	538
19.2. Exposing services using RMI	539
Exporting the service using the RmiServiceExporter	539
Linking in the service at the client	540
19.3. Using Hessian or Burlap to remotely call services via HTTP	541
Wiring up the DispatcherServlet for Hessian and co.	541
Exposing your beans by using the HessianServiceExporter	541
Linking in the service on the client	542
Using Burlap	542
Applying HTTP basic authentication to a service exposed through Hessian or Burlap	543
19.4. Exposing services using HTTP invokers	543
Exposing the service object	543
Linking in the service at the client	544
19.5. Web services	545
Exposing servlet-based web services using JAX-RPC	546
Accessing web services using JAX-RPC	546
Registering JAX-RPC Bean Mappings	548
Registering your own JAX-RPC Handler	549
Exposing servlet-based web services using JAX-WS	549
Exporting standalone web services using JAX-WS	550
Exporting web services using the JAX-WS RI's Spring support	551
Accessing web services using JAX-WS	551
19.6. JMS	552
Server-side configuration	553
Client-side configuration	554
19.7. Auto-detection is not implemented for remote interfaces	555
19.8. Considerations when choosing a technology	555
19.9. Accessing RESTful services on the Client	556
RestTemplate	556
Dealing with request and response headers	558
HTTP Message Conversion	559
StringHttpMessageConverter	559
FormHttpMessageConverter	559
ByteArrayMessageConverter	560
MarshallingHttpMessageConverter	560
MappingJacksonHttpMessageConverter	560
SourceHttpMessageConverter	560
BufferedImageHttpMessageConverter	560
20. Enterprise JavaBeans (EJB) integration	561
20.1. Introduction	561
20.2. Accessing EJBs	561
Concepts	561
Accessing local SLSBs	562

Accessing remote SLSBs	563
Accessing EJB 2.x SLSBs versus EJB 3 SLSBs	564
20.3. Using Spring's EJB implementation support classes	564
EJB 2.x base classes	564
EJB 3 injection interceptor	566
21. JMS (Java Message Service)	568
21.1. Introduction	568
21.2. Using Spring JMS	568
JmsTemplate	568
Connections	569
Caching Messaging Resources	570
SingleConnectionFactory	570
CachingConnectionFactory	570
Destination Management	570
Message Listener Containers	571
SimpleMessageListenerContainer	572
DefaultMessageListenerContainer	572
Transaction management	572
21.3. Sending a Message	573
Using Message Converters	574
SessionCallback and ProducerCallback	575
21.4. Receiving a message	575
Synchronous Reception	575
Asynchronous Reception - Message-Driven POJOs	575
The SessionAwareMessageListener interface	576
The MessageListenerAdapter	576
Processing messages within transactions	578
21.5. Support for JCA Message Endpoints	579
21.6. JMS Namespace Support	581
22. JMX	586
22.1. Introduction	586
22.2. Exporting your beans to JMX	586
Creating an MBeanServer	587
Reusing an existing MBeanServer	588
Lazy-initialized MBeans	589
Automatic registration of MBeans	589
Controlling the registration behavior	589
22.3. Controlling the management interface of your beans	591
The MBeanInfoAssembler Interface	591
Using Source-Level Metadata (JDK 5.0 annotations)	591
Source-Level Metadata Types	593
The AutodetectCapableMBeanInfoAssembler interface	595
Defining management interfaces using Java interfaces	596
Using MethodNameBasedMBeanInfoAssembler	597
22.4. Controlling the ObjectNames for your beans	597

Reading ObjectNames from Properties	598
Using the MetadataNamingStrategy	599
The <context:mbean-export> element	599
22.5. JSR-160 Connectors	600
Server-side Connectors	600
Client-side Connectors	601
JMX over Burlap/Hessian/SOAP	601
22.6. Accessing MBeans via Proxies	602
22.7. Notifications	602
Registering Listeners for Notifications	602
Publishing Notifications	606
22.8. Further Resources	607
23. JCA CCI	608
23.1. Introduction	608
23.2. Configuring CCI	608
Connector configuration	608
ConnectionFactory configuration in Spring	609
Configuring CCI connections	610
Using a single CCI connection	610
23.3. Using Spring's CCI access support	611
Record conversion	611
The CciTemplate	612
DAO support	613
Automatic output record generation	614
Summary	614
Using a CCI Connection and Interaction directly	615
Example for CciTemplate usage	616
23.4. Modeling CCI access as operation objects	618
MappingRecordOperation	618
MappingCommAreaOperation	619
Automatic output record generation	619
Summary	619
Example for MappingRecordOperation usage	620
Example for MappingCommAreaOperation usage	622
23.5. Transactions	623
24. Email	625
24.1. Introduction	625
24.2. Usage	625
Basic MailSender and SimpleMailMessage usage	626
Using the JavaMailSender and the MimeMessagePreparator	627
24.3. Using the JavaMail MimeMessageHelper	628
Sending attachments and inline resources	628
Attachments	628
Inline resources	628
Creating email content using a templating library	629

A Velocity-based example	630
25. Task Execution and Scheduling	632
25.1. Introduction	632
25.2. The Spring TaskExecutor abstraction	632
TaskExecutor types	632
Using a TaskExecutor	634
25.3. The Spring TaskScheduler abstraction	635
The Trigger interface	635
Trigger implementations	636
TaskScheduler implementations	636
25.4. The Task Namespace	637
The 'scheduler' element	637
The 'executor' element	637
The 'scheduled-tasks' element	638
25.5. Annotation Support for Scheduling and Asynchronous Execution	639
The @Scheduled Annotation	639
The @Async Annotation	640
The <annotation-driven> Element	641
25.6. Using the OpenSymphony Quartz Scheduler	641
Using the JobDetailBean	641
Using the MethodInvokingJobDetailFactoryBean	642
Wiring up jobs using triggers and the SchedulerFactoryBean	643
25.7. Using JDK Timer support	644
Creating custom timers	644
Using the MethodInvokingTimerTaskFactoryBean	644
Wrapping up: setting up the tasks using the TimerFactoryBean	645
26. Dynamic language support	646
26.1. Introduction	646
26.2. A first example	646
26.3. Defining beans that are backed by dynamic languages	648
Common concepts	648
The <lang:language/> element	649
Refreshable beans	649
Inline dynamic language source files	652
Understanding Constructor Injection in the context of dynamic-language-backed beans	652
JRuby beans	653
Groovy beans	655
Customising Groovy objects via a callback	657
BeanShell beans	658
26.4. Scenarios	659
Scripted Spring MVC Controllers	659
Scripted Validators	660
26.5. Bits and bobs	661
AOP - advising scripted beans	661

Scoping	661
26.6. Further Resources	662
VII. Appendices	663
A. Classic Spring Usage	664
A.1. Classic ORM usage	664
Hibernate	664
The HibernateTemplate	664
Implementing Spring-based DAOs without callbacks	665
JDO	666
JdoTemplate and JdoDaoSupport	666
JPA	667
JpaTemplate and JpaDaoSupport	667
A.2. Classic Spring MVC	668
A.3. JMS Usage	669
JmsTemplate	669
Asynchronous Message Reception	669
Connections	670
Transaction Management	670
B. Classic Spring AOP Usage	671
B.1. Pointcut API in Spring	671
Concepts	671
Operations on pointcuts	672
AspectJ expression pointcuts	672
Convenience pointcut implementations	672
Static pointcuts	672
Dynamic pointcuts	674
Pointcut superclasses	674
Custom pointcuts	675
B.2. Advice API in Spring	675
Advice lifecycles	675
Advice types in Spring	675
Interception around advice	675
Before advice	676
Throws advice	677
After Returning advice	678
Introduction advice	679
B.3. Advisor API in Spring	682
B.4. Using the ProxyFactoryBean to create AOP proxies	682
Basics	682
JavaBean properties	683
JDK- and CGLIB-based proxies	684
Proxying interfaces	685
Proxying classes	687
Using 'global' advisors	687
B.5. Concise proxy definitions	688

B.6. Creating AOP proxies programmatically with the ProxyFactory	689
B.7. Manipulating advised objects	689
B.8. Using the "autoproxy" facility	691
Autoproxy bean definitions	691
BeanNameAutoProxyCreator	691
DefaultAdvisorAutoProxyCreator	692
AbstractAdvisorAutoProxyCreator	693
Using metadata-driven auto-proxying	693
B.9. Using TargetSources	696
Hot swappable target sources	696
Pooling target sources	697
Prototype target sources	698
ThreadLocal target sources	698
B.10. Defining new Advice types	699
B.11. Further resources	699
C. XML Schema-based configuration	701
C.1. Introduction	701
C.2. XML Schema-based configuration	702
Referencing the schemas	702
The util schema	703
<util:constant/>	703
<util:property-path/>	705
<util:properties/>	706
<util:list/>	707
<util:map/>	708
<util:set/>	708
The jee schema	709
<jee:jndi-lookup/> (simple)	709
<jee:jndi-lookup/> (with single JNDI environment setting)	710
<jee:jndi-lookup/> (with multiple JNDI environment settings)	710
<jee:jndi-lookup/> (complex)	711
<jee:local-slsb/> (simple)	711
<jee:local-slsb/> (complex)	711
<jee:remote-slsb/>	712
The lang schema	712
The jms schema	713
The tx (transaction) schema	713
The aop schema	714
The context schema	714
<property-placeholder/>	715
<annotation-config/>	715
<component-scan/>	715
<load-time-weaver/>	715
<spring-configured/>	715
<mbean-export/>	715

The tool schema	716
The beans schema	716
D. Extensible XML authoring	717
D.1. Introduction	717
D.2. Authoring the schema	717
D.3. Coding a NamespaceHandler	719
D.4. Coding a BeanDefinitionParser	719
D.5. Registering the handler and the schema	720
'META-INF/spring.handlers'	721
'META-INF/spring.schemas'	721
D.6. Using a custom extension in your Spring XML configuration	721
D.7. Meatier examples	722
Nesting custom tags within custom tags	722
Custom attributes on 'normal' elements	725
D.8. Further Resources	727
E. spring-beans-2.0.dtd	728
F. spring.tld	739
F.1. Introduction	739
F.2. The bind tag	739
F.3. The escapeBody tag	740
F.4. The hasBindErrors tag	740
F.5. The htmlEscape tag	740
F.6. The message tag	741
F.7. The nestedPath tag	741
F.8. The theme tag	742
F.9. The transform tag	742
F.10. The url tag	743
F.11. The eval tag	743
G. spring-form.tld	745
G.1. Introduction	745
G.2. The checkbox tag	745
G.3. The checkboxes tag	747
G.4. The errors tag	749
G.5. The form tag	750
G.6. The hidden tag	752
G.7. The input tag	752
G.8. The label tag	754
G.9. The option tag	756
G.10. The options tag	757
G.11. The password tag	758
G.12. The radiobutton tag	760
G.13. The radiobuttons tag	762
G.14. The select tag	764
G.15. The textarea tag	766

Part I. Overview of Spring Framework

The Spring Framework is a lightweight solution and a potential one-stop-shop for building your enterprise-ready applications. However, Spring is modular, allowing you to use only those parts that you need, without having to bring in the rest. You can use the IoC container, with Struts on top, but you can also use only the [Hibernate integration code](#) or the [JDBC abstraction layer](#). The Spring Framework supports declarative transaction management, remote access to your logic through RMI or web services, and various options for persisting your data. It offers a full-featured [MVC framework](#), and enables you to integrate [AOP](#) transparently into your software.

Spring is designed to be non-intrusive, meaning that your domain logic code generally has no dependencies on the framework itself. In your integration layer (such as the data access layer), some dependencies on the data access technology and the Spring libraries will exist. However, it should be easy to isolate these dependencies from the rest of your code base.

This document is a reference guide to Spring Framework features. If you have any requests, comments, or questions on this document, please post them on the user mailing list or on the support forums at <http://forum.springsource.org/>.

1. Introduction to Spring Framework

Spring Framework is a Java platform that provides comprehensive infrastructure support for developing Java applications. Spring handles the infrastructure so you can focus on your application.

Spring enables you to build applications from “plain old Java objects” (POJOs) and to apply enterprise services non-invasively to POJOs. This capability applies to the Java SE programming model and to full and partial Java EE.

Examples of how you, as an application developer, can use the Spring platform advantage:

- Make a Java method execute in a database transaction without having to deal with transaction APIs.
- Make a local Java method a remote procedure without having to deal with remote APIs.
- Make a local Java method a management operation without having to deal with JMX APIs.
- Make a local Java method a message handler without having to deal with JMS APIs.

1.1 Dependency Injection and Inversion of Control

Background

“*The question is, what aspect of control are [they] inverting?*” Martin Fowler posed this question about Inversion of Control (IoC) on his site in 2004. Fowler suggested renaming the principle to make it more self-explanatory and came up with *Dependency Injection*.

For insight into IoC and DI, refer to Fowler's article at <http://martinfowler.com/articles/injection.html>.

Java applications -- a loose term that runs the gamut from constrained applets to n-tier server-side enterprise applications -- typically consist of objects that collaborate to form the application proper. Thus the objects in an application have *dependencies* on each other.

Although the Java platform provides a wealth of application development functionality, it lacks the means to organize the basic building blocks into a coherent whole, leaving that task to architects and developers. True, you can use design patterns such as *Factory*, *Abstract Factory*, *Builder*, *Decorator*, and *Service Locator* to compose the various classes and object instances that make up an application. However, these patterns are simply that: best practices given a name, with a description of what the pattern does, where to apply it, the problems it addresses, and so forth. Patterns are formalized best practices that *you must implement yourself* in your application.

The Spring Framework *Inversion of Control* (IoC) component addresses this concern by providing a

formalized means of composing disparate components into a fully working application ready for use. The Spring Framework codifies formalized design patterns as first-class objects that you can integrate into your own application(s). Numerous organizations and institutions use the Spring Framework in this manner to engineer robust, *Maintainable* applications.

1.2 Modules

The Spring Framework consists of features organized into about 20 modules. These modules are grouped into Core Container, Data Access/Integration, Web, AOP (Aspect Oriented Programming), Instrumentation, and Test, as shown in the following diagram.

Overview of the Spring Framework

Core Container

The *Core Container* consists of the Core, Beans, Context, and Expression Language modules.

The *Core and Beans* modules provide the fundamental parts of the framework, including the IoC and Dependency Injection features. The BeanFactory is a sophisticated implementation of the factory pattern. It removes the need for programmatic singletons and allows you to decouple the configuration and specification of dependencies from your actual program logic.

The *Context* module builds on the solid base provided by the *Core and Beans* modules: it is a means to access objects in a framework-style manner that is similar to a JNDI registry. The Context module inherits its features from the Beans module and adds support for internationalization (using, for example, resource bundles), event-propagation, resource-loading, and the transparent creation of contexts by, for example, a servlet container. The Context module also supports Java EE features such as EJB, JMX ,and basic remoting. The ApplicationContext interface is the focal point of the Context module.

The *Expression Language* module provides a powerful expression language for querying and manipulating an object graph at runtime. It is an extension of the unified expression language (unified EL) as specified in the JSP 2.1 specification. The language supports setting and getting property values, property assignment, method invocation, accessing the context of arrays, collections and indexers, logical and arithmetic operators, named variables, and retrieval of objects by name from Spring's IoC container. It also supports list projection and selection as well as common list aggregations.

Data Access/Integration

The *Data Access/Integration* layer consists of the JDBC, ORM, OXM, JMS and Transaction modules.

The *JDBC* module provides a JDBC-abstraction layer that removes the need to do tedious JDBC coding and parsing of database-vendor specific error codes.

The [ORM](#) module provides integration layers for popular object-relational mapping APIs, including [JPA](#), [JDO](#), [Hibernate](#), and [iBatis](#). Using the ORM package you can use all of these O/R-mapping frameworks in combination with all of the other features Spring offers, such as the simple declarative transaction management feature mentioned previously.

The [OXM](#) module provides an abstraction layer that supports Object/XML mapping implementations for JAXB, Castor, XMLBeans, JiBX and XStream.

The Java Messaging Service ([JMS](#)) module contains features for producing and consuming messages.

The [Transaction](#) module supports programmatic and declarative transaction management for classes that implement special interfaces and for *all your POJOs (plain old Java objects)*.

Web

The *Web* layer consists of the Web, Web-Servlet, Web-Struts, and Web-Portlet modules.

Spring's *Web* module provides basic web-oriented integration features such as multipart file-upload functionality and the initialization of the IoC container using servlet listeners and a web-oriented application context. It also contains the web-related parts of Spring's remoting support.

The *Web-Servlet* module contains Spring's model-view-controller ([MVC](#)) implementation for web applications. Spring's MVC framework provides a clean separation between domain model code and web forms, and integrates with all the other features of the Spring Framework.

The *Web-Struts* module contains the support classes for integrating a classic Struts web tier within a Spring application. Note that this support is now deprecated as of Spring 3.0. Consider migrating your application to Struts 2.0 and its Spring integration or to a Spring MVC solution.

The *Web-Portlet* module provides the MVC implementation to be used in a portlet environment and mirrors the functionality of Web-Servlet module.

AOP and Instrumentation

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

The separate *Aspects* module provides integration with AspectJ.

The *Instrumentation* module provides class instrumentation support and classloader implementations to be used in certain application servers.

Test

The *Test* module supports the testing of Spring components with JUnit or TestNG. It provides consistent loading of Spring ApplicationContexts and caching of those contexts. It also provides mock objects that you can use to test your code in isolation.

1.3 Usage scenarios

The building blocks described previously make Spring a logical choice in many scenarios, from applets to full-fledged enterprise applications that use Spring's transaction management functionality and web framework integration.

Typical full-fledged Spring web application

Spring's declarative transaction management features make the web application fully transactional, just as it would be if you used EJB container-managed transactions. All your custom business logic can be implemented with simple POJOs and managed by Spring's IoC container. Additional services include support for sending email and validation that is independent of the web layer, which lets you choose where to execute validation rules. Spring's ORM support is integrated with JPA, Hibernate, JDO and iBatis; for example, when using Hibernate, you can continue to use your existing mapping files and standard Hibernate SessionFactory configuration. Form controllers seamlessly integrate the web-layer with the domain model, removing the need for ActionForms or other classes that transform HTTP parameters to values for your domain model.

Spring middle-tier using a third-party web framework

Sometimes circumstances do not allow you to completely switch to a different framework. The Spring Framework does *not* force you to use everything within it; it is not an *all-or-nothing* solution. Existing front-ends built with WebWork, Struts, Tapestry, or other UI frameworks can be integrated with a Spring-based middle-tier, which allows you to use Spring transaction features. You simply need to wire up your business logic using an ApplicationContext and use a WebApplicationContext to integrate your web layer.

Remoting usage scenario

When you need to access existing code through web services, you can use Spring's Hessian-, Burlap-, Rmi - or JaxRpcProxyFactory classes. Enabling remote access to existing applications is not difficult.

EJBs - Wrapping existing POJOs

The Spring Framework also provides an access and abstraction layer for Enterprise JavaBeans, enabling you to reuse your existing POJOs and wrap them in stateless session beans for use in scalable, fail-safe web applications that might need declarative security.

Dependency Management and Naming Conventions

Dependency management and dependency injection are different things. To get those nice features of Spring into your application (like dependency injection) you need to assemble all the libraries needed (jar files) and get them onto your classpath at runtime, and possibly at compile time. These dependencies are not virtual components that are injected, but physical resources in a file system (typically). The process of dependency management involves locating those resources, storing them and adding them to classpaths. Dependencies can be direct (e.g. my application depends on Spring at runtime), or indirect (e.g. my application depends on commons-dbc which depends on commons-pool). The indirect dependencies are also known as "transitive" and it is those dependencies that are hardest to identify and manage.

If you are going to use Spring you need to get a copy of the jar libraries that comprise the pieces of Spring that you need. To make this easier Spring is packaged as a set of modules that separate the dependencies as much as possible, so for example if you don't want to write a web application you don't need the spring-web modules. To refer to Spring library modules in this guide we use a shorthand naming convention `spring-*` or `spring-*.jar`, where "*" represents the short name for the module (e.g. `spring-core`, `spring-webmvc`, `spring-jms`, etc.). The actual jar file name that you use may be in this form (see below) or it may not, and normally it also has a version number in the file name (e.g. `spring-core-3.0.0.RELEASE.jar`).

In general, Spring publishes its artifacts to four different places:

- On the community download site <http://www.springsource.org/downloads/community>. Here you find all the Spring jars bundled together into a zip file for easy download. The names of the jars here since version 3.0 are in the form `org.springframework.*-<version>.jar`.
- Maven Central, which is the default repository that Maven queries, and does not require any special configuration to use. Many of the common libraries that Spring depends on also are available from Maven Central and a large section of the Spring community uses Maven for dependency management, so this is convenient for them. The names of the jars here are in the form `spring-*-<version>.jar` and the Maven groupId is `org.springframework`.
- The Enterprise Bundle Repository (EBR), which is run by SpringSource and also hosts all the libraries that integrate with Spring. Both Maven and Ivy repositories are available here for all Spring jars and their dependencies, plus a large number of other common libraries that people use in applications with Spring. Both full releases and also milestones and development snapshots are deployed here. The names of the jar files are in the same form as the community download (`org.springframework.*-<version>.jar`), and the dependencies are also in this "long" form, with external libraries (not from SpringSource) having the prefix `com.springsource`. See the [FAQ](#) for more information.
- In a public Maven repository hosted on Amazon S3 for development snapshots and milestone releases (a copy of the final releases is also held here). The jar file names are in the same form as Maven Central, so this is a useful place to get development versions of Spring to use with other libraries