THE EXPERT'S VOICE® IN SPRING

FOURTH EDITION

Pro Spring

YOUR INDUSTRY STANDARD GUIDE TO THE SPRING FRAMEWORK 4

Chris Schaefer, Clarence Ho, and Rob Harrop

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Covering version 4 of the Spring Framework, this is the most comprehensive Spring reference and practical guide available for harnessing the power of this leading enterprise Java application development framework.

This edition covers core Spring and its integration with other leading Java technologies, such as Hibernate, JPA 2, and WebSocket. We share our insights and real-world experiences with enterprise application development, including remoting, transactions, the web and presentation tiers, and much more.

- With Pro Spring 4, you'll learn how to do the following:
- Use Inversion of Control (IoC) and Dependency Injection (DI)
- Use aspect-oriented programming (AOP) techniques with Spring and learn why they're important
- Build Spring-based web applications using Spring MVC and WebSocket
- Utilize the new Java 8 lambda syntax
- Work with scripting languages like Groovy to provide enhanced functionality for your applications

Arm yourself with the power to build complex Spring applications, from top to bottom. This book is for experienced Java developers who may be learning Spring for the first time or have minimal exposure to the Spring Framework. It's aimed at those who are active in or plan on getting into enterprise Java application development.

CHAPTER 1

Introducing Spring

When we think of the community of Java developers, we are reminded of the hordes of gold rush prospectors of the late 1840s, frantically panning the rivers of North America, looking for fragments of gold. As Java developers, our rivers run rife with open source projects, but, like the prospectors, finding a useful project can be time-consuming and arduous.

A common gripe with many open source Java projects is that they are conceived merely out of the need to fill the gap in the implementation of the latest buzzword-heavy technology or pattern. Having said that, many high-quality, usable projects meet and address a real need for real applications, and in the course of this book, you will meet a subset of these projects. You will get to know one in particular rather well—Spring.

Throughout this book, you will see many applications of different open source technologies, all of which are unified under the Spring Framework. When working with Spring, an application developer can use a large variety of open source tools, without needing to write reams of code and without coupling his application too closely to any particular tool.

In this chapter, as its title indicates, we introduce you to the Spring Framework, rather than presenting any solid examples or explanations. If you are already familiar with the Spring project, you might want to skip this chapter and proceed straight to Chapter 2.

What Is Spring?

Perhaps one the hardest parts of explaining Spring is classifying exactly what it is. Typically, *Spring* is described as a lightweight framework for building Java applications, but that statement brings up two interesting points.

First, you can use Spring to build any application in Java (for example, stand-alone, web, or Java Enterprise Edition (JEE) applications), unlike many other frameworks (such as Apache Struts, which is limited to web applications).

Second, the *lightweight* part of the description doesn't really refer to the number of classes or the size of the distribution, but rather defines the principle of the Spring philosophy as a whole—that is, minimal impact. Spring is lightweight in the sense that you have to make few, if any, changes to your application code to gain the benefits of the Spring core, and should you choose to stop using Spring at any point, you will find doing so quite simple.

Notice that we qualified that last statement to refer to the Spring core only—many of the extra Spring components, such as data access, require a much closer coupling to the Spring Framework. However, the benefits of this coupling are quite clear, and throughout the book we present techniques for minimizing the impact this has on your application.

Evolution of the Spring Framework

The Spring Framework originated from the book *Expert One-on-One: J2EE Design and Development* by Rod Johnson (Wrox, 2002). Over the last decade, the Spring Framework has grown dramatically in core functionality, associated projects, and community support. With the new major release of the Spring Framework, it's worthwhile to take a quick look back at important features that have come along with each milestone release of Spring, leading up to Spring Framework 4.0:

- Spring 0.9
  - The first public release of the framework, based on the book *Expert One-on-One: J2EE Design and Development*
- Spring 1.*x*
  - Spring Core: Bean container and supporting utilities
  - Spring Context: `ApplicationContext`, UI, validation, JNDI, Enterprise JavaBeans (EJB), remoting, and mail support
  - Spring DAO: Transaction infrastructure, Java Database Connectivity (JDBC) and data access object (DAO) support
  - Spring ORM: Hibernate, iBATIS and Java Data Objects (JDO) support
  - Spring AOP: An AOP Alliance–compliant aspect-oriented programming (AOP) implementation
  - Spring Web: Basic integration features such as multipart functionality, context initialization through servlet listeners, and a web-oriented application context
  - Spring Web MVC: Web-based Model-View-Controller (MVC) framework
- Spring 2.*x*
  - Easier XML configuration through use of the new XML Schema–based configuration rather than the DTD format. Notable areas of improvement include bean definitions, AOP, and declarative transactions.
  - New bean scopes for web and portal usage (request, session, and global session)
  - `@AspectJ` annotation support for AOP development
  - Java Persistence API (JPA) abstraction layer
  - Full support for asynchronous JMS message-driven POJOs (for *plain old Java objects*)
  - JDBC simplifications including `SimpleJdbcTemplate` when using Java 5+
  - JDBC named parameter support (`NamedParameterJdbcTemplate`)
  - Form tag library for Spring MVC
  - Introduction of the Portlet MVC framework
  - Dynamic language support: beans can be written in JRuby, Groovy, and BeanShell
  - Notification support and controllable MBean registration in JMX
  - `TaskExecutor` abstraction introduced for scheduling of tasks
  - Java 5 annotation support, specifically for `@Transactional`, `@Required`, in addition to `@AspectJ`

- Spring 2.5.*x*
  - New configuration annotation `@Autowired` and support for JSR-250 annotations (`@Resource`, `@PostConstruct`, `@PreDestroy`)
  - New stereotype annotations: `@Component`, `@Repository`, `@Service`, `@Controller`
  - Auto classpath scanning support to automatically detect and wire classes annotated with stereotype annotations
  - AOP updates: introduction of the `bean(...)` pointcut element and AspectJ load-time weaving
  - Full WebSphere transaction management support
  - In addition to the Spring MVC `@Controller` annotation, `@RequestMapping`, `@RequestParam`, and `@ModelAttribute` annotations added to support request handling through annotation configuration
  - Tiles 2 support
  - JSF 1.2 support
  - JAX-WS 2.0/2.1 support
  - Introduction of the Spring TestContext Framework, providing annotation-driven and integration testing support, agnostic of the testing framework being used
  - Ability to deploy a Spring application context as a JCA adapter
- Spring 3.0.*x*
  - Support for Java 5 features such as generics, varargs, and other improvements
  - First-class support for `Callables`, `Futures`, `ExecutorService` adapters, and `ThreadFactory` integration
  - Framework modules now managed separately with one source-tree per module JAR
  - Introduction of the Spring Expression Language (SpEL)
  - Integration of core JavaConfig features and annotations
  - General-purpose type-conversion system and field-formatting system
  - Comprehensive REST support
  - New MVC XML namespace and additional annotations such as `@CookieValue` and `@RequestHeaders` for Spring MVC
  - Validation enhancements and JSR-303 (“Bean Validation”) support
  - Early support for Java EE 6: `@Async/@Asynchronous` annotation, JSR-303, JSF 2.0, JPA 2.0, and so on
  - Support for embedded databases such as HSQL, H2, and Derby
- Spring 3.1.*x*
  - New Cache abstraction
  - Bean definition profiles can be defined in XML as well as support for the `@Profile` annotation

    - Environment abstraction for unified property management
    - Annotation equivalents for common Spring XML namespace elements such as `@ComponentScan`, `@EnableTransactionManagement`, `@EnableCaching`, `@EnableWebMvc`, `@EnableScheduling`, `@EnableAsync`, `@EnableAspectJAutoProxy`, `@EnableLoadTimeWeaving`, and `@EnableSpringConfigured`
    - Support for Hibernate 4
    - Spring TestContext Framework support for `@Configuration` classes and bean definition profiles
    - `c:` namespace for simplified constructor injection
    - Support for Servlet 3 code-based configuration of the Servlet container
    - Ability to bootstrap the JPA `EntityManagerFactory` without `persistence.xml`
    - Flash and `RedirectAttributes` added to Spring MVC, allowing attributes to survive a redirect by using the HTTP session
    - URI template variable enhancements
    - Ability to annotate Spring MVC `@RequestBody` controller method arguments with `@Valid`
    - Ability to annotate Spring MVC controller method arguments with the `@RequestPart` annotation

- Spring 3.2.*x*
    - Support for Servlet 3–based asynchronous request processing
    - New Spring MVC test framework
    - New Spring MVC annotations `@ControllerAdvice`, `@MatrixVariable`
    - Support for generic types in `RestTemplate` and in `@RequestBody` arguments
    - Jackson JSON 2 support
    - Support for Tiles 3
    - `@RequestBody` or an `@RequestPart` argument can now be followed by an `Errors` argument, making it possible to handle validation errors
    - Ability to exclude URL patterns by using the MVC namespace and JavaConfig configuration options
    - Support for `@DateTimeFormat` without Joda Time
    - Global date and time formatting
    - Concurrency refinements across the framework, minimizing locks and generally improving concurrent creation of scoped/prototyped beans
    - New Gradle-based build system
    - Migration to GitHub: https://github.com/SpringSource/spring-framework
    - Refined Java SE 7 / OpenJDK 7 support in the framework and third-party dependencies. CGLIB and ASM are now included as part of Spring. AspectJ 1.7 is supported in addition to 1.6.

- Spring 4.0
  - Improved getting-started experience via a series of Getting Started guides on the new `www.spring.io/guides` website
  - Removal of deprecated packages and methods from the prior Spring 3 version
  - Java 8 support, raising the minimum Java version to 6 update 18
  - Java EE 6 and above is now considered the baseline for Spring Framework 4.0
  - Groovy bean definition DSL allowing bean definitions to be configured via Groovy syntax
  - Core container, testing, and general web improvements
  - WebSocket, SockJS, and STOMP messaging

Inverting Control or Injecting Dependencies?

The core of the Spring Framework is based on the principle of *Inversion of Control (IoC)*. IoC is a technique that externalizes the creation and management of component dependencies. Consider an example in which class `Foo` depends on an instance of class `Bar` to perform some kind of processing. Traditionally, `Foo` creates an instance of `Bar` by using the new operator or obtains one from some kind of factory class. Using the IoC approach, an instance of `Bar` (or a subclass) is provided to `Foo` at runtime by some external process. This behavior, the injection of dependencies at runtime, led to IoC being renamed by Martin Fowler as the much more descriptive *Dependency Injection (DI)*. The precise nature of the dependencies managed by DI is discussed in Chapter 3.

■ **Note** As you will see in Chapter 3, using the term *Dependency Injection* when referring to Inversion of Control is always correct. In the context of Spring, you can use the terms interchangeably, without any loss of meaning.

Spring's DI implementation is based on two core Java concepts: JavaBeans and interfaces. When you use Spring as the DI provider, you gain the flexibility of defining dependency configuration within your applications in different ways (for example, XML files, Java configuration classes, annotations within your code, or the new Groovy bean definition method). JavaBeans (*POJOs*) provide a standard mechanism for creating Java resources that are configurable in a number of ways, such as constructors and setter methods. In Chapter 3, you will see how Spring uses the JavaBean specification to form the core of its DI configuration model; in fact, any Spring-managed resource is referred to as a *bean*. If you are unfamiliar with JavaBeans, refer to the quick primer we present at the beginning of Chapter 3.

Interfaces and DI are technologies that are mutually beneficial. Clearly designing and coding an application to interfaces makes for a flexible application, but the complexity of wiring together an application designed using interfaces is quite high and places an additional coding burden on developers. By using DI, you reduce the amount of code you need to use an interface-based design in your application to almost zero. Likewise, by using interfaces, you can get the most out of DI because your beans can utilize any interface implementation to satisfy their dependency. The use of interfaces also allows Spring to utilize JDK dynamic proxies (Proxy Pattern) to provide powerful concepts such as AOP for crosscutting concerns.

In the context of DI, Spring acts more like a container than a framework—providing instances of your application classes with all the dependencies they need—but it does so in a much less intrusive way. Using Spring for DI relies on nothing more than following the JavaBeans naming conventions within your classes—there are no special classes from which to inherit or proprietary naming schemes to follow. If anything, the only change you make in an application that uses DI is to expose more properties on your JavaBeans, thus allowing more dependencies to be injected at runtime.

Evolution of Dependency Injection

In the past few years, thanks to the popularity gained by Spring and other DI frameworks, DI has gained wide acceptance among Java developer communities. At the same time, developers were convinced that using DI was a best practice in application development, and the benefits of using DI were also well understood.

The popularity of DI was acknowledged when the Java Community Process (JCP) adopted JSR-330, "Dependency Injection for Java" in 2009. JSR-330 had become a formal Java Specification Request, and as you might expect, one of the specification leads was Rod Johnson—the founder of the Spring Framework.

In JEE 6, JSR-330 became one of the included specifications of the entire technology stack. In the meantime, the EJB architecture (starting from version 3.0) was also revamped dramatically; it adopted the DI model in order to ease the development of various Enterprise JavaBeans apps.

Although we leave the full discussion of DI until Chapter 3, it is worth taking a look at the benefits of using DI rather than a more traditional approach:

- *Reduced glue code*: One of the biggest plus points of DI is its ability to dramatically reduce the amount of code you have to write to glue the components of your application together. Often this code is trivial, so creating a dependency involves simply creating a new instance of an object. However, the glue code can get quite complex when you need to look up dependencies in a JNDI repository or when the dependencies cannot be invoked directly, as is the case with remote resources. In these cases, DI can really simplify the glue code by providing automatic JNDI lookup and automatic proxying of remote resources.
- *Simplified application configuration*: By adopting DI, you can greatly simplify the process of configuring an application. You can use a variety of options to configure those classes that were injectable to other classes. You can use the same technique to express the dependency requirements to the "injector" for injecting the appropriate bean instance or property. In addition, DI makes it much simpler to swap one implementation of a dependency for another. Consider the case where you have a DAO component that performs data operations against a PostgreSQL database and you want to upgrade to Oracle. Using DI, you can simply reconfigure the appropriate dependency on your business objects to use the Oracle implementation rather than the PostgreSQL one.
- *Ability to manage common dependencies in a single repository*: Using a traditional approach to dependency management of common services—for example, data source connection, transaction, and remote services—you create instances (or lookup from some factory classes) of your dependencies where they are needed (within the dependent class). This will cause the dependencies to spread across the classes in your application, and changing them can prove problematic. When you use DI, all the information about those common dependencies is contained in a single repository, making the management of dependencies much simpler and less error prone.
- *Improved testability*: When you design your classes for DI, you make it possible to replace dependencies easily. This is especially handy when you are testing your application. Consider a business object that performs some complex processing; for part of this, it uses a DAO to access data stored in a relational database. For your test, you are not interested in testing the DAO; you simply want to test the business object with various sets of data. In a traditional approach, whereby the business object is responsible for obtaining an instance of the DAO itself, you have a hard time testing this, because you are unable to easily replace the DAO implementation with a mock implementation that returns your test data sets. Instead, you need to make sure your test database contains the correct data and uses the full DAO implementation for your tests. Using DI, you can create a mock implementation of the DAO object that returns the test data sets, and then you can pass this to your business object for testing. This mechanism can be extended for testing any tier of your application and is especially useful for testing web components where you can create mock implementations of `HttpServletRequest` and `HttpServletResponse`.

- *Fostering of good application design*: Designing for DI means, in general, designing against interfaces. A typical injection-oriented application is designed so that all major components are defined as interfaces, and then concrete implementations of these interfaces are created and hooked together using the DI container. This kind of design was possible in Java before the advent of DI and DI-based containers such as Spring, but by using Spring, you get a whole host of DI features for free, and you are able to concentrate on building your application logic, not a framework to support it.

As you can see from this list, DI provides a lot of benefits for your application, but it is not without its drawbacks. In particular, DI can make it difficult for someone not intimately familiar with the code to see just what implementation of a particular dependency is being hooked into which objects. Typically, this is a problem only when developers are inexperienced with DI; after becoming more experienced and following good DI coding practice (for example, putting all injectable classes within each application layer into the same package), developers will be able to discover the whole picture easily. For the most part, the massive benefits far outweigh this small drawback, but you should consider this when planning your application.

Beyond Dependency Injection

The Spring core alone, with its advanced DI capabilities, is a worthy tool, but where Spring really excels is in its myriad of additional features, all elegantly designed and built using the principles of DI. Spring provides features for all layers of an application, from helper application programming interfaces (APIs) for data access right through to advanced MVC capabilities. What is great about these features in Spring is that, although Spring often provides its own approach, you can easily integrate them with other tools in Spring, making these tools first-class members of the Spring family.

Support for Java 8

Java 8 brings many exciting features that Spring Framework 4 supports, most notably lambda expressions and method references with Spring's callback interfaces. Other Java 8 functionality includes first-class support for `java.time` (JSR-310) and parameter name discovery. While Spring Framework 4.0 supports Java 8, compatibility is still maintained back to JDK 6 update 18. The use of a more recent version of Java such as 7 or 8 is recommended for new development projects.

Aspect-Oriented Programming with Spring

AOP provides the ability to implement *crosscutting logic*—that is, logic that applies to many parts of your application—in a single place and to have that logic applied across your application automatically.

Spring's approach to AOP is creating *dynamic proxies* to the target objects and *weaving* the objects with the configured advice to execute the crosscutting logic. By the nature of JDK dynamic proxies, target objects must implement an interface declaring the method in which the AOP advice will be applied.

Another popular AOP library is the Eclipse AspectJ project (`www.eclipse.org/aspectj`), which provides more-powerful features including object construction, class loading, and stronger crosscutting capability.

However, the good news for Spring and AOP developers is that starting from version 2.0, Spring offers much tighter integration with AspectJ. The following are some highlights:

- Support for AspectJ-style pointcut expressions
- Support for `@AspectJ` annotation style, while still using Spring AOP for weaving
- Support for aspects implemented in AspectJ for DI
- Support for load-time weaving within the Spring `ApplicationContext`

■ **Note** Starting with Spring Framework version 3.2, `@AspectJ` annotation support can be enabled with Java Configuration.

Both kinds of AOP have their place, and in most cases, Spring AOP is sufficient for addressing an application's crosscutting requirements. However, for more-complicated requirements, AspectJ can be used, and both Spring AOP and AspectJ can be mixed in the same Spring-powered application.

AOP has many applications. A typical one given in many of the traditional AOP examples involves performing some kind of logging, but AOP has found uses well beyond the trivial logging applications. Indeed, within the Spring Framework itself, AOP is used for many purposes, particularly in transaction management. Spring AOP is covered in full detail in Chapter 5, where we show you typical uses of AOP within the Spring Framework and your own applications, as well as AOP performance and areas where traditional technologies are better suited than AOP.

Spring Expression Language

Expression Language (EL) is a technology to allow an application to manipulate Java objects at runtime. However, the problem with EL is that different technologies provide their own EL implementations and syntaxes. For example, Java Server Pages (JSP) and Java Server Faces (JSF) both have their own EL, and their syntaxes are different. To solve the problem, the Unified Expression Language (EL) was created.

Because the Spring Framework is evolving so quickly, there is a need for a standard expression language that can be shared among all the Spring Framework modules as well as other Spring projects. Consequently, starting in version 3.0, Spring introduced the *Spring Expression Language (SpEL)*. SpEL provides powerful features for evaluating expressions and for accessing Java objects and Spring beans at runtime. The result can be used in the application or injected into other JavaBeans.

Validation in Spring

Validation is another large topic in any kind of application. The ideal scenario is that the validation rules of the attributes within JavaBeans containing business data can be applied in a consistent way, regardless of whether the data manipulation request is initiated from the front end, a batch job, or remotely (or example Web Services, RESTful Web Services, or Remote Procedure Call (RPC)).

To address these concerns, Spring provides a built-in validation API by way of the `Validator` interface. This interface provides a simple, yet concise mechanism allowing you to encapsulate your validation logic into a class responsible for validating the target object. In addition to the target object, the validate method takes an `Errors` object, which is used to collect any validation errors that may occur.

Spring also provides a handy utility class, `ValidationUtils`, which provides convenience methods for invoking other validators, checking for common problems such as empty strings, and reporting errors back to the provided `Errors` object.

Driven by need, the JCP also developed the "Bean Validation" API specification (JSR-303), which provides a standard way of defining bean validation rules. For example, when applying the `@NotNull` annotation to a bean's property, it mandates that the attribute shouldn't contain a null value before being able to persist into the database.

Starting in version 3.0, Spring provides out-of-the-box support for JSR-303. To use the API, just declare a `LocalValidatorFactoryBean` and inject the `Validator` interface into any Spring-managed beans. Spring will resolve the underlying implementation for you. By default, Spring will first look for the Hibernate Validator (`hibernate.org/subprojects/validator`), which is a popular JSR-303 implementation. Many front-end technologies (for example, JSF 2 and Google Web Toolkit), including Spring MVC, also support the application of JSR-303 validation in the user interface. The time when developers needed to program the same validation logic in both the user interface and the back-end layer is gone. The details are discussed in Chapter 10.

■ **Note** Starting with Spring Framework version 4.0, the 1.1 version of the Bean Validation API specification (JSR-349) is supported.

Accessing Data in Spring

Data access and persistence seem to be the most discussed topics in the Java world. Spring provides excellent integration with a choice selection of these data access tools. In addition, Spring makes plain vanilla JDBC a viable option for many projects, with its simplified wrapper APIs around the standard API.

Spring's data access module provides out-of-the-box support for JDBC, Hibernate, JDO, and the JPA.

■ **Note** Starting with Spring Framework version 4.0, iBATIS support has been removed. The MyBatis-Spring project provides integration with Spring, and more information can be found at http://mybatis.github.io/spring/.

However, in the past few years, because of the explosive growth of the Internet and cloud computing, besides relational databases, a lot of other "special-purpose" databases were developed. Examples include databases based on key-value pairs to handle extremely large volumes of data (generally referred to as NoSQL), graph databases, and document databases. To help developers support those databases and to not complicate the Spring data access module, a separate project called Spring Data (http://projects.spring.io/spring-data) was created. The project was further split into different categories to support more-specific database access requirements.

■ **Note** Spring's support of nonrelational databases is not covered in this book. If you are interested in this topic, the Spring Data project mentioned earlier is a good place to look. The project page details the nonrelational databases that it supports, with links to those databases' home pages.

The JDBC support in Spring makes building an application on top of JDBC a realistic undertaking, even for more-complex applications. The support for Hibernate, JDO, and JPA makes already simple APIs even simpler, thus easing the burden on developers. When using the Spring APIs to access data via any tool, you are able to take advantage of Spring's excellent transaction support. You'll find a full discussion of this in Chapter 9.

One of the nicest features in Spring is the ability to easily mix and match data access technologies within an application. For instance, you may be running an application with Oracle, using Hibernate for much of your data access logic. However, if you want to take advantage of some Oracle-specific features, it is simple to implement that part of your data access tier by using Spring's JDBC APIs.

Object/XML Mapping in Spring

Most applications need to integrate or provide services to other applications. One common requirement is to exchange data with other systems, either on a regular basis or in real time. In terms of data format, XML is the most commonly used. As a result, you will often need to transform a JavaBean into XML format, and vice versa.

Spring supports many common Java-to-XML mapping frameworks and, as usual, eliminates the need for directly coupling to any specific implementation. Spring provides common interfaces for marshalling (transforming JavaBeans into XML) and unmarshalling (transforming XML into Java objects) for DI into any Spring beans. Common libraries such as Java Architecture for XML Binding (JAXB), Castor, XStream, JiBX, and XMLBeans are supported. In Chapter 12, when we discuss remotely accessing a Spring application for business data in XML format, you will see how to use Spring's Object/XML Mapping (OXM) support in your application.

Managing Transactions

Spring provides an excellent abstraction layer for transaction management, allowing for programmatic and declarative transaction control. By using the Spring abstraction layer for transactions, you can make it simple to change the underlying transaction protocol and resource managers. You can start with simple, local, resource-specific transactions and move to global, multiresource transactions without having to change your code.

Transactions are covered in full detail in Chapter 9.

Simplifying and Integrating with JEE

With the growing acceptance of DI frameworks such as Spring, a lot of developers have chosen to construct applications by using DI frameworks in favor of the JEE's EJB approach. As a result, the JCP communities also realize the complexity of EJB. Starting in version 3.0 of the EJB specification, the API was simplified, so it now embraces many of the concepts from DI.

However, for those applications that were built on EJB or need to deploy the Spring-based applications in a JEE container and utilize the application server's enterprise services (for example, Java Transaction API (JTA) Transaction Manager, data source connection pooling, and JMS connection factories), Spring also provides simplified support for those technologies. For EJB, Spring provides a simple declaration to perform the JNDI lookup and inject into Spring beans. On the reverse side, Spring also provides simple annotation for injecting Spring beans into EJBs.

For any resources stored in a JNDI-accessible location, Spring allows you to do away with the complex lookup code and have JNDI-managed resources injected as dependencies into other objects at runtime. As a side effect of this, your application becomes decoupled from JNDI, allowing you more scope for code reuse in the future.

MVC in the Web Tier

Although Spring can be used in almost any setting, from the desktop to the Web, it provides a rich array of classes to support the creation of web-based applications. Using Spring, you have maximum flexibility when you are choosing how to implement your web front end.

For developing web applications, the MVC pattern is the most popular practice. In recent versions, Spring has gradually evolved from a simple web framework into a full-blown MVC implementation.

First, view support in Spring MVC is extensive. In addition to standard support for JSP and Java Standard Tag Library (JSTL), which is greatly bolstered by the Spring tag libraries, you can take advantage of fully integrated support for Apache Velocity, FreeMarker, Apache Tiles, and XSLT. In addition, you will find a set of base view classes that make it simple to add Microsoft Excel, PDF, and JasperReports output to your applications.

In many cases, you will find Spring MVC sufficient for your web application development needs. However, Spring can also integrate with other popular web frameworks such as Struts, JSF, Atmosphere, Google Web Toolkit (GWT), and so on.

In the past few years, the technology of web frameworks has evolved quickly. Users have required more-responsive and interactive experiences, and that has resulted in the rise of Ajax as a widely adopted technology in developing rich Internet applications (RIAs). On the other hand, users also want to be able to access their applications from any device, including smartphones and tablets. This creates a need for web frameworks that support HTML5, JavaScript, and CSS3. In Chapter 16, we discuss developing web applications by using Spring MVC.

WebSocket Support

Starting with Spring Framework 4.0, support for the Java API for WebSocket (JSR-356) is available. WebSocket defines an API for creating a persistent connection between a client and server, typically implemented in web browsers and servers. WebSocket-style development opens the door for efficient, full-duplex communication enabling real-time message exchanges for highly responsive applications. Use of WebSocket support is detailed further in Chapter 17.

Remoting Support

Accessing or exposing remote components in Java has never been the simplest of jobs. Using Spring, you can take advantage of extensive support for a wide range of remoting techniques to quickly expose and access remote services.

Spring provides support for a variety of remote access mechanisms, including Java Remote Method Invocation (RMI), JAX-WS, Caucho Hessian and Burlap, JMS, Advanced Message Queuing Protocol (AMQP), and REST. In addition to these remoting protocols, Spring also provides its own HTTP-based invoker that is based on standard Java serialization. By applying Spring's dynamic proxying capabilities, you can have a proxy to a remote resource injected as a dependency into one of your classes, thus removing the need to couple your application to a specific remoting implementation and also reducing the amount of code you need to write for your application. We discuss remote support in Spring in Chapter 12.

Mail Support

Sending e-mail is a typical requirement for many kinds of applications and is given first-class treatment within the Spring Framework. Spring provides a simplified API for sending e-mail messages that fits nicely with the Spring DI capabilities. Spring supports the standard JavaMail API.

Spring provides the ability to create a prototype message in the DI container and uses this as the base for all messages sent from your application. This allows for easy customization of mail parameters such as the subject and sender address. In addition, for customizing the message body, Spring integrates with template engines, such as Apache Velocity, which allow the mail content to be externalized from the Java code.

Job Scheduling Support

Most nontrivial applications require some kind of scheduling capability. Whether this is for sending updates to customers or performing housekeeping tasks, the ability to schedule code to run at a predefined time is an invaluable tool for developers.

Spring provides scheduling support that can fulfill most common scenarios. A task can be scheduled either for a fixed interval or by using a Unix cron expression.

On the other hand, for task execution and scheduling, Spring integrates with other scheduling libraries as well. For example, in the application server environment, Spring can delegate execution to the CommonJ library that is used by many application servers. For job scheduling, Spring also supports libraries including the JDK Timer API and Quartz, a commonly used open source scheduling library.

The scheduling support in Spring is covered in full in Chapter 11.

Dynamic Scripting Support

Starting with JDK 6, Java introduced dynamic language support, in which you can execute scripts written in other languages in a JVM environment. Examples include Groovy, JRuby, and JavaScript.

Spring also supports the execution of dynamic scripts in a Spring-powered application, or you can define a Spring bean that was written in a dynamic scripting language and injected into other JavaBeans. Spring-supported dynamic scripting languages include Groovy, JRuby, and BeanShell. In Chapter 14, we discuss the support of dynamic scripting in Spring in detail.

Simplified Exception Handling

One area where Spring really helps reduce the amount of repetitive, boilerplate code you need to write is in exception handling. The core of the Spring philosophy in this respect is that checked exceptions are overused in Java and that a framework should not force you to catch any exception from which you are unlikely to be able to recover—a point of view that we agree with wholeheartedly.

In reality, many frameworks are designed to reduce the impact of having to write code to handle checked exceptions. However, many of these frameworks take the approach of sticking with checked exceptions but artificially reducing the granularity of the exception class hierarchy. One thing you will notice with Spring is that because of the convenience afforded to the developer from using unchecked exceptions, the exception hierarchy is remarkably granular.

Throughout the book, you will see examples in which the Spring exception-handling mechanisms can reduce the amount of code you have to write and, at the same time, improve your ability to identify, classify, and diagnose errors within your application.

The Spring Project

One of the most endearing things about the Spring project is the level of activity present in the community and the amount of cross-pollination between Spring and other projects such as CGLIB, Apache Geronimo, and AspectJ. One of the most touted benefits of open source is that if the project folded tomorrow, you would be left with the code; but let's face it—you do not want to be left with a code base the size of Spring to support and improve. For this reason, it is comforting to know how well established and active the Spring community is.

Origins of Spring

As noted earlier in this chapter, the origins of Spring can be traced back to *Expert One-to-One: J2EE Design and Development*. In this book, Rod Johnson presented his own framework, called the Interface 21 Framework, which he developed to use in his own applications. Released into the open source world, this framework formed the foundation of the Spring Framework as we know it today.

Spring proceeded quickly through the early beta and release candidate stages, and the first official 1.0 release was made available March 24, 2004. Since then, Spring has undergone dramatic growth, and at the time of this writing, the latest major version of Spring Framework is 4.0.

The Spring Community

The Spring community is one of the best in any open source project we have encountered. The mailing lists and forums are always active, and progress on new features is usually rapid. The development team is truly dedicated to making Spring the most successful of all the Java application frameworks, and this shows in the quality of the code that is reproduced.

As we mentioned already, Spring also benefits from excellent relationships with other open source projects, a fact that is extremely beneficial when you consider the large amount of dependency the full Spring distribution has.

From a user's perspective, perhaps one of the best features of Spring is the excellent documentation and test suite that accompany the distribution. Documentation is provided for almost all the features of Spring, making it easy for new users to pick up the framework. The test suite Spring provides is impressively comprehensive—the development team writes tests for everything. If they discover a bug, they fix that bug by first writing a test that highlights the bug and then getting the test to pass.

Fixing bugs and creating new features is not limited just to the development team! You can contribute code through pull requests against any portfolio of Spring projects through the official GitHub repositories (`http://github.com/spring-projects`). Additionally, issues can be created and tracked by way of the official Spring JIRA (`https://jira.springsource.org/secure/Dashboard.jspa`).

What does all this mean to you? Well, put simply, it means you can be confident in the quality of the Spring Framework and confident that, for the foreseeable future, the Spring development team will continue to improve what is already an excellent framework.

The Spring Tool Suite

To ease the development of Spring-based applications in Eclipse, Spring created the Spring IDE project. Soon after that, SpringSource, the company behind Spring founded by Rod Johnson, created an integrated tool called the Spring Tool Suite (STS), which can be downloaded from www.spring.io/tools. Although it used to be a paid-for product, the tool is now freely available. The tool integrates the Eclipse IDE, Spring IDE, Mylyn (a task-based development environment in Eclipse), Maven for Eclipse, AspectJ Development Tools, and many other useful Eclipse plug-ins into a single package. In each new version, more features are being added, such as Groovy scripting language support, a graphical Spring configuration editor, visual development tools for projects such as Spring Batch and Spring Integration, and support for the Pivotal tc Server application server.

■ **Note** SpringSource was bought by VMWare and incorporated into Pivotal Software, Inc.

In addition to the Java-based suite, a Groovy/Grails Tool Suite is available with similar capabilities but targeted at Groovy and Grails development (www.spring.io/tools).

The Spring Security Project

The Spring Security project (http://projects.spring.io/spring-security), formerly known as the Acegi Security System for Spring, is another important project within the Spring portfolio. Spring Security provides comprehensive support for both web application and method-level security. It tightly integrates with the Spring Framework and other commonly used authentication mechanisms, such as HTTP basic authentication, form-based login, X.509 certificate, and single sign-on (SSO) products (for example, CA SiteMinder). It provides role-based access control to application resources, and in applications with more-complicated security requirements (for example, data segregations), use of an access control list (ACL) is supported. However, Spring Security is mostly used in securing web applications, which we discuss in detail in Chapter 16.

Spring Batch and Integration

Needless to say, batch job execution and integration are common use cases in applications. To cope with this need and to make it easy for developers in these areas, Spring created the Spring Batch and Spring Integration projects. Spring Batch provides a common framework and various policies for batch job implementation, reducing a lot of boilerplate code. By implementing the Enterprise Integration Patterns (EIP), Spring Integration can make integrating Spring applications with external systems easy. We discuss the details in Chapter 20.

Many Other Projects

We've covered the core modules of Spring and some of the major projects within the Spring portfolio, but there are many other projects that have been driven by the need of the community for different requirements. Some examples include Spring Boot, Spring XD, Spring for Android, Spring Mobile, Spring Social, and Spring AMQP. Some of these projects are discussed further in Chapter 20. For additional details, you can refer to the Spring by Pivotal web site (www.spring.io/projects).

Alternatives to Spring

Going back to our previous comments on the number of open source projects, you should not be surprised to learn that Spring is not the only framework offering Dependency Injection features or full end-to-end solutions for building applications. In fact, there are almost too many projects to mention. In the spirit of being open, we include a brief discussion of several of these frameworks here, but it is our belief that none of these platforms offers quite as comprehensive a solution as that available in Spring.

JBoss Seam Framework

Founded by Gavin King (the creator of the Hibernate ORM library), the Seam Framework (`www.seamframework.org`) is another full-blown DI-based framework. It supports web application front-end development (JSF), business logic layer (EJB 3), and JPA for persistence. As you can see, the main difference between Seam and Spring is that the Seam Framework is built entirely on JEE standards. JBoss also contributes the ideas in the Seam Framework back to the JCP and has become JSR-299, "Contexts and Dependency Injection for the Java EE Platform" (CDI).

Google Guice

Another popular DI framework is Google Guice (`http://code.google.com/p/google-guice`). Led by the search engine giant Google, Guice is a lightweight framework that focuses on providing DI for application configuration management. It was also the reference implementation of JSR-330, "Dependency Injection for Java".

PicoContainer

PicoContainer (`http://picocontainer.com`) is an exceptionally small DI container that allows you to use DI for your application without introducing any dependencies other than PicoContainer. Because PicoContainer is nothing more than a DI container, you may find that as your application grows, you need to introduce another framework, such as Spring, in which case you would have been better off using Spring from the start. However, if all you need is a tiny DI container, then PicoContainer is a good choice, but since Spring packages the DI container separately from the rest of the framework, you can just as easily use that and keep the flexibility for the future.

JEE 7 Container

As discussed previously, the concept of DI was widely adopted and also realized by JCP. When you are developing an application for application servers compliant with JEE 7 (JSR-342) , you can use standard DI techniques across all layers.

Summary

In this chapter, we gave you a high-level view of the Spring Framework, complete with discussions of all the major features, and we guided you to the relevant sections of the book where these features are discussed in detail. After reading this chapter, you should understand what Spring can do for you; all that remains is to see *how* it can do it.

In the next chapter, we discuss all the information you need to know to get up and running with a basic Spring application. We show you how to obtain the Spring Framework and discuss the packaging options, the test suite, and the documentation. Also, Chapter 2 introduces some basic Spring code, including the time-honored "Hello World!" example in all its DI-based glory.

CHAPTER 2

Getting Started

Often the hardest part of coming to grips with any new development tool is figuring out where to begin. Typically, this problem is worse when the tool offers as many choices as Spring. Fortunately, getting started with Spring isn't that hard if you know where to look first. In this chapter, we present you with all the basic knowledge you need to get off to a flying start. Specifically, you will look at the following:

- *Obtaining Spring:* The first logical step is to obtain or build the Spring JAR files. If you want to get up and running quickly, simply use the dependency management snippets in your build system with the provided examples located at `http://projects.spring.io/spring-framework`. However, if you want to be on the cutting edge of Spring development, check out the latest version of the source code from Spring's GitHub repository (`http://github.com/spring-projects/spring-framework`).
- *Spring packaging options:* Spring packaging is modular; it allows you to pick and choose which components you want to use in your application and to include only those components when you are distributing your application. Spring has many modules, but you need only a subset of these modules depending on your application's needs. Each module has its compiled binary code in a JAR file along with corresponding Javadoc and source JARs.
- *Spring guides:* The new Spring web site includes a Guides section located at `www.spring.io/guides`. The guides are meant to be quick, hands-on instructions for building the "Hello World" of any development task with Spring. These guides also reflect the latest Spring project releases and techniques, providing you with the most up-to-date samples available.
- *Test suite and documentation:* One of the things members of the Spring community are most proud of is their comprehensive test suite and documentation set. Testing is a big part of what the team does. The documentation set provided with the standard distribution is also excellent.
- *Putting a spring into "Hello World"* All bad punning aside, we think the best way to get started with any new programming tool is to dive right in and write some code. We present a simple example, which is a full DI-based implementation of everyone's favorite, "Hello World!" Don't be alarmed if you don't understand all the code right away; full discussions follow later in the book.

If you are already familiar with the basics of the Spring Framework, feel free to proceed straight to Chapter 3 to dive into IoC and DI in Spring. However, even if you are familiar with the basics of Spring, you may find some of the discussions in this chapter interesting, especially those on packaging and dependencies.

Obtaining the Spring Framework

Before you can get started with any Spring development, you need to obtain the Spring code. You have a couple of options for retrieving the code: you can use your build system to bring in the modules you would like to use, or you can check out and build the code from the Spring GitHub repository. Using a dependency management tool such as Maven or Gradle is often the most straightforward approach, as all you need to do is declare the dependency in the configuration file, and let the tool obtain the required libraries for you.

Quick Start

Visit the Spring Framework project page (`http://projects.spring.io/spring-framework`) to obtain a dependency management snippet for your build system to include the latest-release RELEASE version of Spring in your project. You can also use milestones/nightly snapshots for upcoming releases or previous versions.

Checking Spring Out of GitHub

If you want to get a grip on new features before they make their way even into the snapshots, you can check out the source code directly from Spring by Pivotal's GitHub repository. To check out the latest version of the Spring code, first install Git, which you can download from `http://git-scm.com/` Then open a terminal shell and run the following command:

```
git clone git://github.com/spring-projects/spring-framework.git
```

> See the `README.md` file in the project root for full details and requirements on how to build from source.

Understanding Spring Packaging

Spring modules are simply JAR files that package the required code for that module. After you understand the purpose of each module, you can then select the modules required in your project and include them in your code.

Understanding Spring Modules

As of Spring version 4.0.2.RELEASE, Spring comes with 20 modules, packaged into 20 JAR files. Table 2-1 describes these JAR files and their corresponding modules. The actual JAR file format is, for example, `spring-aop-4.0.2.RELEASE.jar`, though we have included only the specific module portion for simplicity (as in `aop`, for example).

Table 2-1. *Spring Modules*

JAR File	Description
`aop`	This module contains all the classes you need to use Spring's AOP features within your application. You also need to include this JAR in your application if you plan to use other features in Spring that use AOP, such as declarative transaction management. Moreover, classes that support integration with AspectJ are packed in this module too.
`aspects`	This module contains all the classes for advanced integration with the AspectJ AOP library. For example, if you are using Java classes for your Spring configuration and need AspectJ-style annotation-driven transaction management, you will need this module.
`beans`	This module contains all the classes for supporting Spring's manipulation of Spring beans. Most of the classes here support Spring's bean factory implementation. For example, the classes required for processing the Spring XML configuration file and Java annotations were packed into this module.
`context`	This module contains classes that provide many extensions to the Spring core. You will find that all classes need to use Spring's `ApplicationContext` feature (covered in Chapter 5), along with classes for EJB, Java Naming and Directory Interface (JNDI), and Java Management Extensions (JMX) integration. Also contained in this module are the Spring remoting classes, classes for integration with dynamic scripting languages (for example, JRuby, Groovy, and BeanShell), the Bean Validation (JSR-303) API, scheduling and task execution, and so on.
`context-support`	This module contains further extensions to the `spring-context` module. On the user-interface side, there are classes for mail support and integration with templating engines such as Velocity, FreeMarker, and JasperReports. Also, integration with various task execution and scheduling libraries including CommonJ and Quartz are packaged here.
`core`	This is the core module that you will need for every Spring application. In this JAR file, you will find all the classes that are shared among all other Spring modules (for example, classes for accessing configuration files). Also, in this JAR, you will find selections of extremely useful utility classes that are used throughout the Spring code base and that you can use in your own application.
`expression`	This module contains all support classes for Spring Expression Language (SpEL).
`instrument`	This module includes Spring's instrumentation agent for Java Virtual Machine (JVM) bootstrapping. This JAR file is required for using load-time weaving with AspectJ in a Spring application.
`instrument-tomcat`	This module includes Spring's instrumentation agent for JVM bootstrapping in the Tomcat server.
`jdbc`	This module includes all classes for JDBC support. You will need this module for all applications that require database access. Classes for supporting data sources, JDBC data types, JDBC templates, native JDBC connections, and so on, are packed in this module.
`jms`	This module includes all classes for JMS support.
`messaging`	This module contains key abstractions taken from the Spring Integration project to serve as a foundation for message-based applications and adds support for STOMP messages.

(*continued*)

Table 2-1. (*continued*)

JAR File	Description
`orm`	This module extends Spring's standard JDBC feature set with support for popular ORM tools including Hibernate, JDO, JPA, and the data mapper iBATIS. Many of the classes in this JAR depend on classes contained in the spring-jdbc JAR file, so you definitely need to include that in your application as well.
`oxm`	This module provides support for Object/XML Mapping (OXM). Classes for abstraction of XML marshalling and unmarshalling and support for popular tools such as Castor, JAXB, XMLBeans, and XStream are packed into this module.
`test`	As we mentioned earlier, Spring provides a set of mock classes to aid in testing your applications. Many of these mock classes are used within the Spring test suite, so they are well tested and make testing your applications much simpler. Certainly we have found great use for the mock `HttpServletRequest` and `HttpServletResponse` classes in unit tests for our web applications. On the other hand, Spring provides a tight integration with the JUnit unit-testing framework, and many classes that support the development of JUnit test cases are provided in this module; for example, the `SpringJUnit4ClassRunner` provides a simple way to bootstrap the Spring `ApplicationContext` in a unit test environment.
`tx`	This module provides all classes for supporting Spring's transaction infrastructure. You will find classes from the transaction abstraction layer to support of the Java Transaction API (JTA) and integration with application servers from major vendors.
`web`	This module contains the core classes for using Spring in your web applications, including classes for loading an `ApplicationContext` feature automatically, file upload support classes, and a bunch of useful classes for performing repetitive tasks such as parsing integer values from the query string.
`webmvc`	This module contains all the classes for Spring's own MVC framework. If you are using a separate MVC framework for your application, you won't need any of the classes from this JAR file. Spring MVC is covered in more detail in Chapter 16.
`web-portlet`	This module provides support for using Spring MVC in developing portlets for deployment to a portal server environment.
`websocket`	This module provides support for the Java API for WebSocket (JSR-356).

■ **Note** You no longer need an explicit dependency on the ASM module, as it is now packaged with the Spring core.

Choosing Modules for Your Application

Without a dependency management tool such as Maven or Gradle, choosing which modules to use in your application may be a bit tricky. For example, if you require Spring's bean factory and DI support only, you still need several modules including `spring-core`, `spring-beans`, `spring-context`, and `spring-aop`. If you need Spring's web application support, you then need to further add `spring-web` and so on. Thanks to build tool features such as Maven's transitive dependencies support, all required third-party libraries would be included automatically.

Accessing Spring Modules on the Maven Repository

Founded by Apache Software Foundation, Maven (`http://maven.apache.org`) has become one of the most popular tools in managing the dependencies for Java applications, from open source to enterprise environments.

Maven is a powerful application building, packaging, and dependency management tool. It manages the entire build cycle of an application, from resource processing and compiling, to testing and packaging. There also exists a large number of Maven plug-ins for various tasks, such as updating databases and deploying a packaged application to a specific server (for example, Tomcat, JBoss, or WebSphere).

Almost all open source projects support distribution of their library via the Maven repository. The most popular one is the Maven Central repository hosted on Apache, and you can access and search for the existence and related information of an artifact on the Maven Central web site (`http://search.maven.org`). If you download and install Maven into your development machine, you automatically gain access to the Maven Central repository. Some other open source communities (for example, JBoss and Spring by Pivotal) also provide their own Maven repository for their users. However, in order to be able to access those repositories, you need to add the repository into your Maven's setting file or in your project's Project Object Model (POM) file.

A detailed discussion of Maven is not in the scope of this book, and you can always refer to the online documentation or books that give you a detailed reference to Maven. However, since Maven is widely adopted, it's worth mentioning the structure of Spring's packaging on the Maven repository.

A group ID, artifact ID, packaging type, and version identify each Maven artifact. For example, for `log4j`, the group ID is `log4j`, the artifact ID is `log4j`, and the packaging type is `jar`. Under that, different versions are defined. For example, for version 1.2.16, the artifact's file name becomes `log4j-1.2.16.jar` under the group ID, artifact ID, and version folder.

Using Spring Documentation

One of the aspects of Spring that makes it such a useful framework for developers who are building real applications is its wealth of well-written, accurate documentation. In every release, the Spring Framework's documentation team works hard to ensure that all the documentation is finished and polished by the development team. This means that every feature of Spring is not only fully documented in the Javadoc but is also covered in the Spring reference manual included in every distribution. If you haven't yet familiarized yourself with the Spring Javadoc and the reference manual, do so now. This book is not a replacement for either of these resources; rather, it is a complementary reference, demonstrating how to build a Spring-based application from the ground up.

Putting a Spring into "Hello World!"

We hope by this point in the book you appreciate that Spring is a solid, well-supported project that has all the makings of a great tool for application development. However, one thing is missing—we haven't shown you any code yet. We are sure you are dying to see Spring in action, and because we cannot go any longer without getting into the code, let's do just that. Do not worry if you do not fully understand all the code in this section; we go into much more detail on all the topics as we proceed through the book.

Building the Sample "Hello World!" Application

Now, we are sure you are familiar with the traditional "Hello World!" example, but just in case you have been living on the moon for the past 30 years, Listing 2-1 shows the Java version in all its glory.

Listing 2-1. Typical "Hello World!" Example

```
package com.apress.prospring4.ch2;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World!");
    }
}
```

As examples go, this one is pretty simple—it does the job, but it is not very extensible. What if we want to change the message? What if we want to output the message differently, maybe to standard error instead of standard output or enclosed in HTML tags rather than as plain text?

We are going to redefine the requirements for the sample application and say that it must support a simple, flexible mechanism for changing the message, and it must be easy to change the rendering behavior. In the basic "Hello World!" example, you can make both of these changes quickly and easily by just changing the code as appropriate. However, in a bigger application, recompiling takes time, and it requires the application to be fully tested again. A better solution is to externalize the message content and read it in at runtime, perhaps from the command-line arguments shown in Listing 2-2.

Listing 2-2. Using Command-Line Arguments with "Hello World!"

```
package com.apress.prospring4.ch2;

public class HelloWorldWithCommandLine {
    public static void main(String[] args) {
        if (args.length > 0) {
            System.out.println(args[0]);
        } else {
            System.out.println("Hello World!");
        }
    }
}
```

This example accomplishes what we wanted—we can now change the message without changing the code. However, there is still a problem with this application: the component responsible for rendering the message is also responsible for obtaining the message. Changing how the message is obtained means changing the code in the renderer. Add to this the fact that we still cannot change the renderer easily; doing so means changing the class that launches the application.

If we take this application a step further (away from the basics of "Hello World!"), a better solution is to refactor the rendering and message retrieval logic into separate components. Plus, if we really want to make our application flexible, we should have these components implement interfaces and define the interdependencies between the components and the launcher using these interfaces.

By refactoring the message retrieval logic, we can define a simple `MessageProvider` interface with a single method, `getMessage()`, as shown in Listing 2-3.

Listing 2-3. The `MessageProvider` Interface

```
package com.apress.prospring4.ch2;

public interface MessageProvider {
    String getMessage();
}
```

In Listing 2-4, the MessageRenderer interface is implemented by all components that can render messages.

Listing 2-4. The MessageRenderer Interface

```
package com.apress.prospring4.ch2;

public interface MessageRenderer {
    void render();
    void setMessageProvider(MessageProvider provider);
    MessageProvider getMessageProvider();
}
```

As you can see, the MessageRenderer interface has a method, render(), and also a JavaBean-style method, setMessageProvider(). Any MessageRenderer implementations are decoupled from message retrieval and delegate that responsibility to the MessageProvider with which they are supplied. Here, MessageProvider is a dependency of MessageRenderer. Creating simple implementations of these interfaces is easy, as shown in Listing 2-5.

Listing 2-5. The HelloWorldMessageProvider Class

```
package com.apress.prospring4.ch2;

public class HelloWorldMessageProvider implements MessageProvider {
    @Override
    public String getMessage() {
        return "Hello World!";
    }
}
```

You can see that we have created a simple MessageProvider that always returns "Hello World!" as the message. The StandardOutMessageRenderer class (shown in Listing 2-6) is just as simple.

Listing 2-6. The StandardOutMessageRenderer Class

```
package com.apress.prospring4.ch2;

public class StandardOutMessageRenderer implements MessageRenderer {
    private MessageProvider messageProvider;

    @Override
    public void render() {
        if (messageProvider == null) {
            throw new RuntimeException(
                "You must set the property messageProvider of class:"
                + StandardOutMessageRenderer.class.getName());
        }

        System.out.println(messageProvider.getMessage());
    }

    @Override
    public void setMessageProvider(MessageProvider provider) {
        this.messageProvider = provider;
    }
```

```
    @Override
    public MessageProvider getMessageProvider() {
        return this.messageProvider;
    }
}
```

Now all that remains is to rewrite the `main()` method of our entry class, as shown in Listing 2-7.

Listing 2-7. Refactored "Hello World!"

```
package com.apress.prospring4.ch2;

public class HelloWorldDecoupled {
    public static void main(String[] args) {
        MessageRenderer mr = new StandardOutMessageRenderer();
        MessageProvider mp = new HelloWorldMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

The code here is fairly simple: we instantiate instances of `HelloWorldMessageProvider` and `StandardOutMessageRenderer`, although the declared types are `MessageProvider` and `MessageRenderer`, respectively. This is because we need to interact only with the methods provided by the interface in the programming logic, and `HelloWorldMessageProvider` and `StandardOutMessageRenderer` already implemented those interfaces, respectively. Then, we pass the `MessageProvider` to the `MessageRenderer` and invoke `MessageRenderer.render()`. If we compile and run this program, we get the expected "Hello World!" output.

Now, this example is more like what we are looking for, but there is one small problem. Changing the implementation of either the `MessageRenderer` or `MessageProvider` interface means a change to the code. To get around this, we can create a simple factory class that reads the implementation class names from a properties file and instantiates them on behalf of the application (see Listing 2-8).

Listing 2-8. The `MessageSupportFactory` Class

```
package com.apress.prospring4.ch2;

import java.io.FileInputStream;
import java.util.Properties;

public class MessageSupportFactory {
    private static MessageSupportFactory instance;

    private Properties props;
    private MessageRenderer renderer;
    private MessageProvider provider;

    private MessageSupportFactory() {
        props = new Properties();

        try {
            props.load(new FileInputStream("com/apress/prospring4/ch2/msf.properties"));
```

```
            String rendererClass = props.getProperty("renderer.class");
            String providerClass = props.getProperty("provider.class");

            renderer = (MessageRenderer) Class.forName(rendererClass).newInstance();
            provider = (MessageProvider) Class.forName(providerClass).newInstance();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }

    static {
        instance = new MessageSupportFactory();
    }

    public static MessageSupportFactory getInstance() {
        return instance;
    }

    public MessageRenderer getMessageRenderer() {
        return renderer;
    }

    public MessageProvider getMessageProvider() {
        return provider;
    }
}
```

The implementation here is trivial and naïve, the error handling is simplistic, and the name of the configuration file is hard-coded, but we already have a substantial amount of code. The configuration file for this class is quite simple:

```
renderer.class=com.apress.prospring4.ch2.StandardOutMessageRenderer
provider.class=com.apress.prospring4.ch2.HelloWorldMessageProvider
```

Make a simple modification to the `main()` method (as shown in Listing 2-9), and we are in business.

Listing 2-9. Using `MessageSupportFactory`

```
package com.apress.prospring4.ch2;

public class HelloWorldDecoupledWithFactory {
    public static void main(String[] args) {
        MessageRenderer mr = MessageSupportFactory.getInstance().getMessageRenderer();
        MessageProvider mp = MessageSupportFactory.getInstance().getMessageProvider();
        mr.setMessageProvider(mp);
        mr.render();
    }
}
```

Before we move on to see how we can introduce Spring into this application, let's quickly recap what we have done. Starting with the simple "Hello World!" application, we defined two additional requirements that the application must fulfill. The first was that changing the message should be simple, and the second was that changing the rendering mechanism should also be simple. To meet these requirements, we introduced two interfaces: `MessageProvider` and `MessageRenderer`. The `MessageRenderer` interface depends on an implementation of the `MessageProvider` interface to be able to retrieve a message to render. Finally, we added a simple factory class to retrieve the names of the implementation classes and instantiate them as applicable.

Refactoring with Spring

The final example shown earlier met the goals we laid out for our sample application, but there are still problems with it. The first problem is that we had to write a lot of glue code to piece the application together, while at the same time keeping the components loosely coupled. The second problem is that we still had to provide the implementation of `MessageRenderer` with an instance of `MessageProvider` manually. We can solve both of these problems by using Spring.

To solve the problem of too much glue code, we can completely remove the `MessageSupportFactory` class from the application and replace it with a Spring interface, `ApplicationContext`. Don't worry too much about this interface; for now, it is enough to know that this interface is used by Spring for storing all the environmental information with regard to an application being managed by Spring. This interface extends another interface, `ListableBeanFactory`, which acts as the provider for any Spring-managed beans instance (see Listing 2-10).

Listing 2-10. Using Spring's `ApplicationContext`

```
package com.apress.prospring4.ch2;

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

public class HelloWorldSpringDI {
    public static void main(String[] args) {
        ApplicationContext ctx = new ClassPathXmlApplicationContext
           ("META-INF/spring/app-context.xml");

        MessageRenderer mr = ctx.getBean("renderer", MessageRenderer.class);
        mr.render();
    }
}
```

In Listing 2-10, you can see that the `main()` method obtains an instance of `ClassPathXmlApplicationContext` (the application configuration information is loaded from the file `META-INF/spring/app-context.xml` in the project's classpath), typed as `ApplicationContext`, and from this, it obtains the `MessageRenderer` instances by using the `ApplicationContext.getBean()` method. Don't worry too much about the `getBean()` method for now; just know that this method reads the application configuration (in this case, an XML file), initializes Spring's `ApplicationContext` environment, and then returns the configured bean instance. This XML file (`app-context.xml`) serves the same purpose as the one we used for `MessageSupportFactory` (see Listing 2-11).

Listing 2-11. Spring XML Application Configuration

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

    <bean id="provider" class="com.apress.prospring4.ch2.HelloWorldMessageProvider"/>

    <bean id="renderer" class="com.apress.prospring4.ch2.StandardOutMessageRenderer"
        p:messageProvider-ref="provider"/>
</beans>
```

The previous file shows a typical Spring `ApplicationContext` configuration. First, Spring's namespaces are declared, and the default namespace is `beans`. The `beans` namespace is used to declare the beans that need to be managed by Spring, and its dependency requirements (for the preceding example, the `renderer` bean's `messageProvider` property is referencing the `provider` bean) for Spring to resolve and inject those dependencies.

Afterward, we declare the bean with the ID `provider` and the corresponding implementation class. When Spring sees this bean definition during `ApplicationContext` initialization, it will instantiate the class and store it with the specified ID.

Then the `renderer` bean is declared, with the corresponding implementation class. Remember that this bean depends on the `MessageProvider` interface for getting the message to render. To inform Spring about the DI requirement, we use the `p` namespace attribute. The tag attribute `p:messageProvider-ref="provider"` tells Spring that the bean's property, `messageProvider`, should be injected with another bean. The bean to be injected into the property should reference a bean with the ID `provider`. When Spring sees this definition, it will instantiate the class, look up the bean's property named `messageProvider`, and inject it with the bean instance with the ID `provider`.

As you can see, upon the initialization of Spring's `ApplicationContext`, the `main()` method now just obtains the `MessageRenderer` bean by using its type-safe `getBean()` method (passing in the ID and the expected return type, which is the `MessageRenderer` interface) and calls `render()`; Spring has created the `MessageProvider` implementation and injected it into the `MessageRenderer` implementation. Notice that we didn't have to make any changes to the classes that are being wired together using Spring. In fact, these classes have no reference to Spring and are completely oblivious to its existence. However, this isn't always the case. Your classes can implement Spring-specified interfaces to interact in a variety of ways with the DI container.

With our new Spring configuration and modified `main()` method, let's see it in action. Using Maven, enter the following commands into your terminal to build the project and the root of your source code:

```
mvn clean package dependency:copy-dependencies
```

The only required Spring module to be declared in your Maven POM is `spring-context`. Maven will automatically bring in any transitive dependencies required for this module. The `dependency:copy-dependencies` goal will copy all required dependencies into a directory called `dependency` in the `target` directory. This path value will also be used as an appending prefix to the library files added to `MANIFEST.MF` when building the JAR. See the Chapter 2 source code (available on the Apress web site), specifically the Maven `pom.xml`, for more information if you are unfamiliar with the Maven JAR building configuration and process.

Finally, to run the Spring DI sample, enter the following commands:

```
cd target ; java -jar hello-world-4.0-SNAPSHOT.jar
```

And at this point, you should see some log statements generated by the Spring container's startup process followed by our expected "Hello World!" output.