Pro Spring 2.5

ISBN-13 (pbk): 978-1-59059-921-1

ISBN-10 (pbk): 1-59059-921-7

ISBN-13 (electronic): 978-1-4302-0506-7

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

SpringSource is the company behind Spring, the de facto standard in enterprise Java. SpringSource is a leading provider of enterprise Java infrastructure software, and delivers enterprise class software, support and services to help organizations utilize Spring. The open source–based Spring Portfolio is a comprehensive enterprise application framework designed on long-standing themes of simplicity and power. With more than five million downloads to date, Spring has become an integral part of the enterprise application infrastructure at organizations worldwide. For more information, visit `www.springsource.com`.

Lead Editors: Steve Anglin and Tom Welsh
Technical Reviewer: Rick Evans
Editorial Board: Clay Andres, Steve Anglin, Ewan Buckingham, Tony Campbell, Gary Cornell, Jonathan Gennick, Kevin Goff, Matthew Moodie, Joseph Ottinger, Jeffrey Pepper, Frank Pohlmann, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Project Manager: Sofia Marchant
Copy Editors: Heather Lang, Damon Larson
Associate Production Director: Kari Brooks-Copony
Production Editor: Kelly Winquist
Compositor: Kinetic Publishing Services
Proofreader: April Eddy
Indexer: Broccoli Information Management
Cover Designer: Kurt Krames
Manufacturing Director: Tom Debolski

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `http://www.apress.com/info/bulksales`.

The source code for this book is available to readers at `http://www.apress.com`. You will need to answer questions pertaining to this book in order to successfully download the code.

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. And yet, more and more developers are turning to open source tools and code. Open source brings innovative code and few restrictions on its usage, allowing developers to focus on the core of the applications they build.

A common gripe about many open source Java projects is that they exist merely to fill a gap in the implementation of the latest buzzword-heavy technology or pattern. Another problem is that some projects lose all their momentum: code that looked very promising in version 0.1 never reaches version 0.2, much less 1.0. Having said that, many high-quality, user-friendly projects meet and address a real need for real applications. In the course of this book, you will meet a carefully chosen subset of these projects. You will get to know one in particular rather well—Spring.

Spring has come a long way since the early code written by Rod Johnson in his book *Expert One-to-One J2EE Design and Development* (Wrox, October 2002). It has seen contributions from the most respected Java developers in the world and reached version 2.5.

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 infrastructure code and without coupling his application too closely to any particular tool. This chapter is a gentle introduction to the Spring Framework. If you are already familiar with Spring, you might want to skip this chapter and go straight to Chapter 2, which deals with the setup and introduces the "Hello, World" application in Spring.

Our main aims in this book are to provide as comprehensive a reference to the Spring Framework as we can and, at the same time, give plenty of practical, application-focused advice without it seeming like a clone of the documentation. To help with this, we build a full application using Spring throughout the book to illustrate how to use Spring technologies.

What Is Spring?

The first thing we need to explain is the name Spring. We will use "Spring" in most of the text of this book, but we may not always mean the same thing. Sometimes, we will mean the Spring Framework and sometimes the Spring project. We believe the distinction will always be clear and that you will not have any trouble understanding our meaning.

The core of the Spring Framework is based on the principle of Inversion of Control (IoC). Applications that follow the IoC principle use configuration that describes the dependencies between its components. It is then up to the IoC framework to satisfy the configured dependencies. The "inversion" means that the application does not control its structure; it is up to the IoC framework to do that.

Consider an example where an instance of class `Foo` depends on an instance of class `Bar` to perform some kind of processing. Traditionally, `Foo` creates an instance of `Bar` using the new operator or obtains one from some kind of factory class. Using the IoC technique, an instance of `Bar` (or a subclass) is provided to `Foo` at runtime by some external process. This injection of dependencies at runtime has sometimes led to IoC being given the much more descriptive name dependency injection (DI). The precise nature of the dependencies managed by DI is discussed in Chapter 3.

Spring's DI implementation puts focus on loose coupling: the components of your application should assume as little as possible about other components. The easiest way to achieve loose coupling in Java is to code to interfaces. Imagine your application's code as a system of components: in a web application, you will have components that handle the HTTP requests and then use the components that contain the business logic of the application. The business logic components, in turn, use the data access objects (DAOs) to persist the data to a database. The important concept is that each component does not know what concrete implementation it is using; it only sees an interface. Because each component of the application is aware only of the interfaces of the other components, we can switch the implementation of the components (or entire groups or layers of components) without affecting the components that use the changed components. Spring's DI core uses the information from your application's configuration files to satisfy the dependencies between its components. The easiest way to allow Spring to set the dependencies is to follow the JavaBean naming conventions in your components, but it is not a strict requirement (for a quick introduction to JavaBeans, go to Chapter 3).

When you use DI, you allow dependency configuration to be externalized from your code. JavaBeans provide a standard mechanism for creating Java resources that are configurable in a standard way. 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, take a look at the quick primer at the beginning of Chapter 3.

Interfaces and DI are mutually beneficial. We are sure that everyone reading this book will agree that designing and coding an application to interfaces makes for a flexible application that is much more amenable to unit testing. But the complexity of writing code that manages the dependencies between the components of an application designed using interfaces is quite high and places an additional coding burden on developers. By using DI, you reduce the amount of extra code you need for an interface-based design 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.

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 than, say, the EJB container that allows you to create persistent entity beans. Most importantly, Spring will manage the dependencies between the components of your application automatically. All you have to do is create a configuration file that describes the dependencies; Spring will take care of the rest. Using Spring for DI requires nothing more than following the JavaBeans naming conventions within your classes (a requirement that, as you will see in Chapter 4, you can bypass using Spring's method injection support)—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.

■ **Note** A container builds the environment in which all other software components live. Spring is a container, because it creates the components of your application and the components are children of the container.

A framework is a collection of components that you can use to build your applications. Spring is a framework, because it provides components to build common parts of applications, such as data access support, MVC support, and many others.

Although we leave a 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:

Reduce glue code: One of the biggest plus points of DI is its ability to reduce dramatically the amount of code you have to write to glue the different components of your application together. Often, this code is trivial—where creating a dependency involves simply creating a new instance of a class. 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.

Externalize dependencies: You can externalize the configuration of dependencies, which allows you to reconfigure easily without needing to recompile your application. This gives you two interesting benefits. First, as you will see in Chapter 4, DI in Spring gives you the ideal mechanism for externalizing all the configuration options of your application for free. Second, this externalization of dependencies 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.

Manage dependencies in a single place: In the traditional approach to dependency management, you create instances of your dependencies where they are needed—within the dependent class. Even worse, in typical large applications, you usually use a factory or locator to find the dependent components. That means that your code depends on the factory or locator as well as the actual dependency. In all but the most trivial of applications, you will have dependencies spread across the classes in your application, and changing them can prove problematic. When you use DI, all the information about dependencies is the responsibility of a single component (the Spring IoC container), making the management of dependencies much simpler and less error prone.

Improve testability: When you design your classes for DI, you make it possible to replace dependencies easily. This comes in 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 object to access data stored in a relational database. 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, where the service object is responsible for obtaining an instance of the DAO itself, you have a hard time testing this, because you are unable to replace the DAO implementation easily with a dummy implementation that returns your test data. Instead, you need to make sure that 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 that returns the test data, and then you can pass this to your service 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 fake implementations of `HttpServletRequest` and `HttpServletResponse`.

Foster 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 wired 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 can 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, but it is not without its drawbacks too. In particular, DI can make seeing just what implementation of a particular dependency is being hooked into which objects difficult, especially for someone not intimately familiar with the code. Typically, this is only a problem when developers are inexperienced with DI; after becoming more experienced, developers find that the centralized view of an application given by Spring DI lets them see the whole picture. 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 tools to help build every layer of an application, from helper application programming interfaces (APIs) for data access right through to advanced model-view-controller (MVC) capabilities. What is great about these features is that, although Spring often provides its own approach, you can easily integrate them with other tools, making them all first-class members of the Spring family.

Aspect-Oriented Programming with Spring

Aspect-oriented programming (AOP) is one of the technologies of the moment in the programming space. AOP lets you implement crosscutting logic—that is, logic that applies to many parts of your application—in a single place, and then have that logic automatically applied right across the application. AOP is enjoying an immense amount of time in the limelight at the moment; however, behind all the hype is a truly useful technology that has a place in any Java developer's toolbox.

There are two main kinds of AOP implementation. Static AOP, such as AspectJ (`www.aspectj.org`), provides a compile-time solution for building AOP-based logic and adding it to an application. Dynamic AOP, such as that in Spring, allows crosscutting logic to be applied arbitrarily to any other code at runtime. Finally, in Spring 2.5, you can use load-time dynamic weaving, which applies the crosscutting logic when the class loader loads the class. Both kinds of AOP have their places, and indeed, Spring provides features to integrate with AspectJ. This is covered in more detail in Chapters 5 and 6.

There are many applications for AOP. The typical one given in many traditional examples involves performing some kind of tracing, but AOP has found many more ambitious uses, even within the Spring Framework itself, particularly in transaction management. Spring AOP is covered in depth in Chapters 5–7, where we show you typical uses of AOP within the Spring Framework and your own application. We also look into the issue of performance and consider some areas where traditional technologies can work better than AOP.

Accessing Data in Spring

Data access and persistence seem to be the most often discussed topics in the Java world. It seems that you cannot visit a community site such as `www.theserverside.com` without being bombarded with articles and blog entries describing the latest, greatest data access tool.

Spring provides excellent integration with a choice selection of these data access tools. Moreover, Spring makes the use of JDBC, a viable option for many projects thanks to its simplified wrapper APIs around the standard JDBC API. As of Spring version 1.1, you have support for JDBC, Hibernate, iBATIS, and Java Data Objects (JDO).

The JDBC support in Spring makes building an application on top of JDBC realistic, even for complex applications. The support for Hibernate, iBATIS, and JDO makes their already simple APIs even simpler, thus easing the burden on developers. When using the Spring APIs to access data via

any tool, you can take advantage of Spring's excellent transaction support. A full discussion of this support can be found in Chapter 15.

One of Spring's nicest features is the ability to mix and match data access technologies easily 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 Oracle-specific features, it is simple to implement that particular part of your data access tier using Spring's JDBC APIs.

Simplifying and Integrating with Java EE

There has been a lot of discussion recently about the complexity of various Java EE APIs, especially those of EJB. It is evident from the EJB 3.0 specification that this discussion has been taken on board by the expert group, and EJB 3.0 brought some simplifications and many new features. Even with the simplifications over EJB 2.0, using Spring's simplified support for many Java EE technologies is still more convenient. For instance, Spring provides a selection of classes for building and accessing EJB resources. These classes cut out a lot of the grunt work from both tasks and provide a more DI-oriented API for 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 code becomes decoupled from JNDI, giving you more scope for code reuse in the future.

As of version 1.0.2, Spring does not support JMS access. However, the CVS repository already contains a large array of classes that are to be introduced in 1.1. Using these classes simplifies all interaction with Java Message Service (JMS) destinations and should reduce a lot of the boilerplate code you need to write in order to use JMS from your Spring applications.

Chapters 11–13 explain how Spring works with the most important Java EE application components; Chapter 14 addresses application integration issues; and Chapter 20 deals with management of Java EE applications.

Job Scheduling Support

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

Spring supports two scheduling mechanisms: one uses the `Timer` class, which has been available since Java 1.3; and the other uses the Quartz scheduling engine. Scheduling based on the `Timer` class is quite primitive and is limited to fixed periods defined in milliseconds. With Quartz, on the other hand, you can build complex schedules using the Unix cron format to define when tasks should be run.

Spring's scheduling support is covered in full in Chapter 11.

Mail Support

Sending e-mail is a typical requirement for many different 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 its DI capabilities. It supports pluggable implementations of the mail API and comes complete with two implementations: one uses JavaMail, and the other uses Jason Hunter's `MailMessage` class from the `com.oreilly.servlet` package available from `http://servlets.com/cos`.

Spring lets you create a prototype message in the DI container and use 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. However, there is no support for customizing the message body outside of the code. In Chapter 12, we look at Spring's mail support in detail and discuss a solution that combines templating engines such as Velocity and FreeMarker and Spring, allowing mail content to be externalized from the Java code.

In addition to simple mail-sending code, we will show how to use Spring events to implement fully asynchronous messaging infrastructure. We will make use of the Spring Java Management Extensions (JMX) features to show how to create an efficient management console for the mail queues.

Dynamic Languages

Dynamic languages in Spring allow you to implement components of your application in languages other than Java (Spring 2.5 supports BeanShell, JRuby, and Groovy). This allows you to externalize part of your application's code so that it can be easily updated by administrators and power users. You could do this even without Spring, but the support built into Spring means that the rest of your application will not be aware that a component is implemented in another language; it will appear to be an ordinary Spring bean.

Many large applications have to deal with complex business processes. This would not be too difficult to handle in Java, but in most cases, these processes change over time, and users want to be able to make the changes themselves. This is where a domain-specific language implementation comes in.

In Chapter 14, you will see how to use the dynamic language support in Spring 2.5 and how to use it to implement a simple domain-specific language.

Remoting Support

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

Spring supports a variety of remote access mechanisms, including Java RMI, JAX-RPC, Caucho Hessian, and Caucho Burlap. In addition to these remoting protocols, Spring 1.1 introduced its own HTTP-based protocol 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 components, 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.

As well as making it easy to access remote components, Spring provides excellent support for exposing a Spring-managed resource as a remote service. This lets you export your service using any of the remoting mechanisms mentioned earlier, without needing any implementation-specific code in your application.

Integrating applications written in different programming languages, and possibly running on different platforms, is one of the most compelling reasons for using remote services. In Chapter 14, we will show how to use remoting between Java applications and a C# Windows rich client application making full use of a Spring service running on a Unix system.

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 easily 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 15.

The Spring MVC Framework

Although Spring can be used in almost any application, from a service-only application, through web and rich-client ones, it provides a rich array of classes for creating web-based applications. Using Spring, you have maximum flexibility when you are choosing how to implement your web front end.

For a web application of any complexity, it makes sense to use a framework with a paradigm that clearly separates the processing logic from the views. To achieve this, you can use the Spring Struts support or use Spring's own excellent MVC framework. To implement complex page flows, you can use Spring Web Flow. You can use a large array of different view technologies, from JavaServer Pages (JSP) and Apache's Jakarta Velocity to Apache POI (to generate Microsoft Excel output) and iText (to create output in Adobe PDF format). The Spring MVC framework is quite comprehensive and provides support classes that address the majority of your requirements. For the rest, you can easily extend the MVC framework to add in your own functionality.

The view support in Spring MVC is extensive and steadily improving. As well as standard support for JSP, which is greatly bolstered by the Spring tag libraries, you can take advantage of fully integrated support for Jakarta Velocity, FreeMarker, Jakarta Tiles (separate from Struts), and XSLT. Moreover, you will find a set of base view classes that make it simple to add Excel and PDF output to your applications.

We cover the Spring MVC implementation in Chapter 16.

Spring Web Flow

Web Flow represents a new way of developing web applications, especially ones that rely on fairly complex transitions between the pages. Web Flow greatly simplifies the implementation of such systems. Because it is a Spring project, it integrates closely with the Spring MVC framework. Just like Spring MVC, Web Flow can use any type of view technology.

We discuss Web Flow in detail in Chapter 18.

AJAX

AJAX is not only a buzzword of Web 2.0 but also an important technique for creating rich web applications. Put simply, it allows web applications to interact with servers without causing unnecessary page reloads. You may argue that it has little to do with the Spring Framework, but if you need to build highly interactive Web 2.0 applications, you cannot escape it. We will show you how to write the necessary code to add AJAX functionality to your Spring web applications. Because Spring does not provide any framework-level infrastructure to deal with the implementation of AJAX web applications, we will show some important design and performance choices you have to make in your web applications.

Chapter 18 covers AJAX applications in much more detail.

Internationalization

Internationalization is an important aspect of any large application; Spring has extensive support for creating multilanguage applications, and we will show you how to deal with the complexities of this task. The issues facing the developers of multilanguage applications are twofold: first, they must write code without any textual information hard-coded; second, they must design the application in a way that will allow for easy translation in the future.

This issue is further highlighted in situations where developers have to deal with exceptions: in most cases, the message of the exception will be in one language—the language of the developer. We will show you how to overcome this limitation.

All aspects of internationalization are covered in Chapter 17.

Simplified Exception Handling

One area where Spring really helps to reduce the amount of repetitive, boilerplate code you need to write is exception handling. The Spring philosophy 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 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 by using unchecked exceptions, the exception hierarchy is remarkably granular. Throughout this book, you will see how Spring's exception-handling mechanisms can reduce the amount of code you have to write, while improving your ability to identify, classify, and diagnose application errors.

The Spring Project

Among the most attractive things about the Spring project are the level of activity currently present in the community and the amount of cross-pollination with other projects, such as CGLIB, Apache Geronimo, and AspectJ. One of the most often touted benefits of open source is that if a project on which you rely folds tomorrow, you would be left with the code. But let's face it—you do not want to be left with a codebase 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 and how many successful applications are using the Spring Framework at their core.

Origins of Spring

As mentioned previously, the origins of Spring can be traced back to the book *Expert One-to-One J2EE Design and Development* by Rod Johnson (Wrox, 2002). In this book, Rod presented his Interface 21 Framework, which he had developed to use in his own applications. Released into the open source world, this framework formed the foundation of Spring 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 several major releases and is currently in its 2.5 release (at the time of this writing).

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 dedicated to making Spring the most successful of all the Java application frameworks, and this shows in the quality of the code that is produced. Much of the ongoing development in Spring consists of reworking existing code to be faster, smaller, neater, or all three.

Spring benefits from excellent relationships with other open source projects, which is a good thing when you consider how much the full Spring distribution relies on integration with other products. From a user's perspective, one of the best things about Spring is the excellent documentation and test suite that come with it. Documentation is provided for almost all Spring features, making it easier for new users to pick up the framework. The test suite is impressively comprehensive, because the development team writes tests for everything. If they find a bug, they fix it by first writing a test that highlights the bug and getting the test to pass.

What does all this mean to you? Well, it means that you can be confident in the quality of the Spring Framework and know that, for the foreseeable future, the Spring development team intends to go on improving what is already an excellent framework.

Spring for Microsoft .NET

The main Spring Framework is 100 percent Java-based. However, due to the success of the Java version, developers in the .NET world started to feel a little bit left out, so Mark Pollack and Rod Johnson started the Spring .NET project. The two projects have completely different development

teams, so the .NET project should have minimal impact on the development of the Java version of the Spring Framework. In fact, the authors believe that this is excellent news. Contrary to popular belief in the Java world, .NET is not a load of garbage produced by the Beast—a fact that we can attest to after delivering several successful .NET applications to our clients.

This project opens up whole new avenues for cross-pollination, especially since .NET already has the lead in some areas, such as source-level metadata, and should help to create better product on both platforms. Another side effect of this project is that it makes the move between platforms much easier for developers, because you can use Spring on both sides. This is all the more true since other projects, such as Hibernate and iBATIS, now have .NET equivalents. You can find more information on Spring .NET at `www.springframework.net`.

The Spring IDE

In all but the simplest Spring applications, the application's configuration files become fairly large and complex; it is convenient to use some kind of integrated development environment (IDE) to help you write the code.

The Spring IDE project is another offshoot of the main Spring project, and it functions as a plug-in for the Eclipse platform. Using Spring IDE, you can get full source highlighting and code insight functionality for your Spring configuration files. You can also reduce the number of errors that can creep into your configuration files, thus speeding up the development cycle. In addition to Spring IDE in Eclipse, you can use IntelliJ IDEA for your Java and Spring development. The Spring support in IntelliJ IDEA 7.0 is indeed excellent.

The Spring Security (Formerly Acegi)

The Spring Security module evolved directly from Acegi, which was a security system built on top of Spring. It provides the full spectrum of security services required for Spring-based applications, including multiple authentication back ends, single sign-on support, and caching. We do not cover Acegi in any detail in this book, but you can find more details at `http://acegisecurity.sourceforge.net/`. Support for Acegi is provided through the Spring forums at `http://forum.springframework.org`.

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 DI or full end-to-end support 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 some of these frameworks here, but we believe that none of them offer quite as comprehensive a solution as Spring.

PicoContainer

PicoContainer (`www.picocontainer.org`) is an exceptionally small (100kB) DI container that allows you to use DI for your application without introducing any dependencies other than PicoContainer itself. 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. If all you need is a tiny DI container, PicoContainer is a good choice. But since Spring packages the DI container separate from the rest of the framework, you can just as easily use that and keep your options open.

NanoContainer

NanoContainer (`www.nanocontainer.org`) is an extension to PicoContainer for managing trees of individual PicoContainer containers. Because Spring provides all the same functionality in the standard DI

container, NanoContainer is not really a major improvement over Spring. Where NanoContainer becomes interesting is in its support for scripting languages that interact with it. However, Spring comes with full support for scripting languages as well.

Keel Framework

The Keel Framework (`www.keelframework.org`) is more of a metaframework, in that most of its capabilities come from other frameworks that are all brought together under a single roof. For instance, DI functionality comes from the Apache Avalon container, while web functionality comes from Struts or a similar framework. Keel has many implementations of the same components and links them all together into a cohesive structure, allowing you to swap out implementations with minimal impact on your application. Despite its wide feature set, Keel does not seem to have enjoyed the same level of acceptance as Spring. Although we have investigated Keel only briefly, we feel that this is partially to do with the level of accessibility. Spring is immediately accessible to developers of all levels, whereas Keel seems to be more complex. Having said that, Keel's feature set is impressive, and it is certainly a direct competitor for Spring.

Google Guice

The Guice (pronounced "juice") framework focuses purely on dependency injection. As such, it is not a direct competition for the Spring Framework; in fact, you can use Spring-managed beans in Guice. Apart from the focus of the framework, the main difference between Guice and Spring is the approach to the configuration of the applications. Guice uses automatic wiring or annotation-based configuration. Automatic wiring means that the framework examines the components it is aware of and will try to guess the dependencies between them. The guess is based on the dependency's type and name. As even the creators of Guice admit (and we wholeheartedly agree), automatic wiring is not suitable for large enterprise applications. For complex applications, Guice authors recommend using annotation-based configuration. Because Guice uses annotations, it does not need any complex configuration files like Spring.

Unfortunately, by adding annotations, you limit the code you write only to Guice. Even with this drawback, Guice is an excellent framework, and its big advantage is that it can be used in cooperation with Spring.

The Sample Code

Throughout the text, we will use two main approaches to sample code. To demonstrate fine and very specific points, we will create single-purpose small applications. To demonstrate complex code, we will create applications that may share code across multiple chapters. You can download the code from Source Code section of the Apress web site (`http://www.apress.com`).

Summary

In this chapter, we presented you with a high-level view of the Spring Framework complete with discussions of all the major features, and we pointed you to the sections of this book where those features are discussed in detail. We also had a very brief look at some of the concepts that we will be dealing with throughout this book. After reading this chapter, you should have an idea of what Spring can do for you; all that remains is to see how it can do it.

In the next chapter, we explain everything 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. On that note, let's press on!