# Pro Spring 2.5

Jan Machacek, Aleksa Vukotic,
Anirvan Chakraborty, and Jessica Ditt

Apress®

**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.

*To Marc, who still thinks that beans grow on trees*
*—Jan*

*To my parents, Andja and Slobodan, who have guided me through life and encouraged me to follow my own path*
*—Aleksa*

*To my parents, Sujata and Kalyan, for their unconditional love and support*
*—Anirvan*

*To Ioannis, for the vital nudges when I needed them most (and so much more), and to my family, just for being my family*
*—Jessica*

# Contents at a Glance



## PART 1 Getting Started with Spring



## PART 2 Data Access



## PART 3 Enterprise Application Components

## PART 4 Java EE 5

# Contents

# PART 2 Data Access

# Foreword

It was with a heavy heart that I made the decision not to participate in writing *Pro Spring 2.5*. I am deeply thankful that Jan was around to pick up this book and run with it. *Pro Spring* has been a big part of my life for over three years, and I didn't relinquish the reins lightly. When Juergen and I set out working on Spring Framework 2.0, I knew that I wouldn't have the time to participate in the writing process and write the software at the same time. Fortunately, Jan was there to step into the breach.

Jan and Apress had additionally planned to release *Pro Spring 2.0*, but Juergen and I inadvertently made it impossible for them to keep up by making many changes to the Spring Framework. I vividly remember cringing when updating all the JSP form tags, knowing that I was creating yet more work for Jan.

With the 2.5 release just on the horizon, Jan made the sensible choice to forego a 2.0 edition and head straight for 2.5. This was a wise move. The Spring Framework 2.5 release reflects the state of the art in both the Spring Framework and in enterprise Java frameworks as a whole. A guide book to this critical tool is necessary reading for any conscientious Java developer.

I recall, back in the early days of running Cake Solutions, when we decided we needed to hire another programmer. We were very inexperienced at hiring in general, and hiring programmers is fraught with problems. We knew that we wanted to get a graduate, but we never imagined that we would get someone as accomplished as Jan.

I remember, in his first week, he wrote a complete desktop mailing package from scratch—and it worked. Over the last five years, Jan has been at the center of most of the projects run at Cake, many of which are large-scale Java products based on the Spring Framework. His knowledge of Spring comes from an immense amount of practical experience: he has been in the trenches with Spring since version 1.0 and has delivered successful systems on top of it.

To his credit, Jan realized that writing *Pro Spring 2.5* was too big a job for just one man, so he roped in the rest of the Cake Solutions team to help him. This prospect excited me greatly— a team of real programmers, with real experience in Spring, passing along that knowledge. There is no doubt that many will find this book to be an indispensable reference.

And so, although I am disappointed at being unable to work on this book myself, I am glad that Jan was there to deliver what so many people have been asking for, an updated version of *Pro Spring*. Enjoy,

Rob Harrop
*Principal Software Engineer and Lead Engineer*
*of the SpringSource Application Platform*

# About the Authors

**JAN MACHACEK** is the chief software architect at Cake Solutions, which places him at the center of all architectural decisions in all projects. Apart from the architecture, Jan is often called on to help with some of the most complex and challenging areas of the implementation. Since joining Cake, Jan has proven his expertise in Java not only by taking on a wide variety of complex projects but also through his collection of published works. In his free time, Jan is a keen cyclist and a member of the Manchester Wheelers' Club. He tries his best in various time trials and road races.

Jan authored Chapters 1–4, 6, 9, 11, 14, 16, and 22.

**ALEKSA VUKOTIC** is a software architect at Cake Solutions. He oversees the architecture as well as the implementation of most of Cake's projects. He has extensive experience with most Java EE technologies, particularly Spring MVC and Security. He also has the knack, which means he can solve virtually any technical problem. He is an excellent tutor and is in charge of directing a team of Cake Solutions developers, helping the team in solving the most complex problems. As well as his interest in Java and .NET platforms, Aleksa enjoys sports, music, and nights out. Aleksa works with Jan on all of the major projects at Cake Solutions.

Aleksa authored Chapters 5, 8, 10, 15, 17, and 21.

**ANIRVAN CHAKRABORTY** is a senior developer at Cake Solutions. His extensive experience with the Spring Framework and attention to detail puts him in charge of the implementation of some of the challenging aspects of Cake Solutions's projects. Anirvan takes great pride in his code and always makes sure the code can be used as an example to others. When he does not have his head buried in Java EE and Linux, he enjoys good food and drink with his friends. He is also an ardent follower of the sport of cricket and enjoys reading detective novels.

Anirvan authored Chapters 7, 13, 19, and 20.

**JESSICA DITT** has been a developer at London-based Avenue A | Razorfish since early 2008. Prior to that, she was part of the Cake Solutions team for 2.5 years. She has been working on numerous enterprise-level projects, all of which were written using the Spring Framework and Spring Web Flow. Jessica has acquired significant expertise in efficient indexing using Lucene and has efficiently addressed Java EE application scalability issues using Gigaspaces. Out of the office, Jessica is a keen volleyball player and enjoys spending time in the gym.

Jessica authored Chapters 12 and 18.

# About the Technical Reviewer

**RICK EVANS** is an independent contractor based in the UK with many years of experience working in the health, financial, and retail sectors. Over the years, Rick has committed on a number of open source projects, including Spring and Spring.NET. A polished teacher and mentor, Rick often speaks professionally about technology and delivers training on a wide range of enterprise technologies and disciplines.

# Acknowledgments

When writing a book, a substantial amount of work goes on behind the scenes, and authors are backed by an excellent team of editors, proofreaders, and technical reviewers. This book was no exception, and we would like to thank everyone who worked on the book. Our thanks goes to Rick, the technical reviewer, who has done enormous amounts of work to ensure that this book is of the highest quality. The great editorial team at Apress also deserves our thanks: most importantly, Sofia Marchant; our editor, Tom Welsh; Heather Lang; Kelly Winquist; and many others. Without their help, we would not have been able to complete this book. I would also like to thank Rob Harrop for agreeing to write the Foreword. Finally, we all wish to thank the managing director of Cake Solutions, Guy Remond; he gave some of Cake Solutions's time to us to work on the book.

# Introduction

Recently, the Java world has witnessed a dramatic shift away from so-called "heavyweight" architectures such as Enterprise JavaBeans (EJB) toward lighter weight frameworks such as Spring. Complex and container-dependent services, such as CMP, and transaction management systems have been replaced with simpler alternatives such as Hibernate and aspect-oriented programming (AOP). At the core, Spring provides a comprehensive, lightweight container based on the principle of Inversion of Control (IoC), on which you can build your own applications. On top of this container, Spring provides a myriad of useful services, bringing together a large range of highly competent open source projects into a single cohesive framework.

The quality of the Spring Framework has seen it replacing traditional Java EE architectures in many cases; as a result, more and more developers see the need for comprehensive Spring skills. Despite Spring having quite an extensive suite of documentation and examples, we feel that many developers are still struggling to understand how to use Spring and, more importantly, how to use it effectively. Because of this, we decided to write a new edition of *Pro Spring*.

At first, we thought we would just update a few chapters and call it *Pro Spring 2.5*. However, we quickly realized that Spring 2.5 brought so many new features and improvements that, although we kept the old *Pro Spring* name, this is a completely new book.

Through this book, you will learn how to use Spring to build better web and stand-alone applications and how to sift through the many choices available to you through the framework. Our aim is to provide you with all the knowledge you need to use Spring *effectively* in your own applications and to give you insight into what is happening behind the scenes in Spring.

For example, you will

- Learn the fundamentals of IoC in the context of AOP.
- Become aware of the seamlessness and power of Spring by referencing the easy-to-understand sample applications we provide.
- Learn how to replace common EJB features with Spring alternatives, including Spring's comprehensive AOP-based transaction management framework.
- Effectively manage your Spring components and applications using Spring's built-in JMX engine.
- Learn how to add scheduling to your Spring application with Quartz.

After reading this book, you will be equipped with all the knowledge you need to build applications effectively using Spring and its related open source projects.

PART 1

# Getting Started with Spring

**In this part of the book, we will introduce the Spring Framework, starting with the steps needed to download the binary and source distributions. We will next explain the Inversion of Control (IoC) pattern and aspect-oriented programming (AOP). IoC and AOP lie at the heart of the Spring Framework. Toward the end of Part 1, we will show you, in full detail, various ways to configure Spring-managed beans. Finally, we will round off Part 1 with a look at some of the most important Spring design patterns.**

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.