Spring Security 3

Secure your web applications against malicious intruders with this easy to follow practical guide

Foreword by Luke Taylor, Spring Security Project Lead

Peter Mularien

Spring Security 3

Secure your web applications against malicious intruders with this easy to follow practical guide

Peter Mularien

Spring Security 3

Credits

Author

Peter Mularien

Reviewers

Scott Battaglia

Carlos Sanchez

Acquisition Editor

Steven Wilding

Development Editor

Neha Patwari

Technical Editors

Meeta Rajani

Sandesh Modhe

Indexer

Hemangini Bari

Editorial Team Leader

Gagandeep Singh

Project Team Leader

Lata Basantani

Project Coordinator

Poorvi Nair

Proofreader

Lesley Harrison

Graphics

Geetanjali Sawant

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

Foreword

Spring Security was started by Ben Alex in 2003, when it was called "The Acegi Security System for Spring" or more commonly just "Acegi Security". Over the years, it developed under Ben's leadership into a sophisticated authentication and access-control system and became widely adopted as the standard solution for securing Spring Framework based applications. In the early days, there was always a small band of contributors scattered about the globe, some of whom came and went, some of whom are still active committers. I first became involved in 2004. Project discussions took place at strange hours of the day, due to the time differences and it was at least a couple of years before I met Ben in person, when he was on a trip to Europe from his native Australia. Now we both work full-time for SpringSource, and Spring Security, which is an official Spring Portfolio project, used in critical applications all over the planet.

Spring Security always had trouble shaking off the reputation of being difficult to learn. It is a "hands-on" framework where people are encouraged to customize or extend the code to fulfill requirements that go beyond the basic out of the box options. Most things are possible, but sometimes an in-depth understanding of the internals is needed to satisfy a requirement, and that understanding mainly comes with experience. The XML namespace configuration options that were introduced in Spring Security 2.0 allow users to get started with simple use cases, but the learning curve can be frustrating for those who want to take advantage of the full power of the framework. This is compounded by the fact that security is a complex subject in general, with a whole new set of unfamiliar concepts and, of course, any software developer involved in securing an application must also be a competent engineer with knowledge of technologies and protocols, which many developers are barely aware of. For anyone faced with a deadline, the prospect of getting up-to-speed can be daunting.

This is the first book dedicated to Spring Security and it will provide all the help you need to get started, illustrated with plenty of concrete practical examples, as well as detailed coverage of more advanced topics such as internal Spring Security architecture, customization, and integration with single sign-on systems such as CAS, OpenID, and Kerberos. Its publication is well-timed as it covers the recent Spring Security 3.0 release, which involved a major restructuring of the project.

Peter is an experienced Spring and Spring Security user, and also a regular in the Spring Security forum where he has answered hundreds of user questions, giving him first hand insight into the kind of practical difficulties that people encounter. Much of that insight has now been distilled into the contents of this book. Very often the difficulties people have are not directly related to Spring Security, but due to problems with other external technologies. You may not be familiar with setting up SSL, understanding LDAP directories or the practical aspects of using different OpenID providers, but in practice you don't care under what subject heading the problem is listed, you just want to solve it as quickly as possible. This is where a book of this kind can really make a difference compared to what you'll typically find in a project reference manual. Peter provides ample coverage of these related topics, supplying just the information you need to understand what's happening and to get the job done.

A lot of work has gone into this book and I think it will prove to be an invaluable resource, both for first-time Spring Security users and those who would like to expand their existing knowledge. What you learn will stand you in good stead for the future. Spring Security is a key member of the Spring Portfolio of Open Source projects and will continue to be highly relevant for Spring developers. What does the future hold for Spring Security? You can expect further integration with existing Spring projects (such as Spring Roo, which is Ben's main focus these days) as well as innovative support for new technologies both in the project core and through the related Security Extensions project (which is discussed in detail in Chapter 12, Spring Security Extensions). And of course you can influence the development of the project yourself by getting involved, submitting patches, or proposing new extension projects.

I wish you all the best with your projects and hope you enjoy using Spring Security.

Luke Taylor

Spring Security Project Lead

About the Author

Peter Mularien is currently a senior developer and architect at a mid-sized enterprise software vendor. He has worked in web-based application development and has been a consultant on a wide range of technologies, offering his services to well-known as well as not so known companies, for over ten years. His Java-and Spring-focused blog—"It's Only Software", remains quite popular for its readability and quick tutorials on topics intended to get developers up to speed quickly on complex technology. Peter is also a recognizable participant in the free Spring Community Forums, ranking in the top 30 of all-time posters.

To my wife and children for your patience and love through late nights and weekends—my deepest and devoted thanks and hugs.

About the Reviewers

Scott Battaglia holds a BS and MS in Computer Science from Rutgers University and is finishing his MBA from Rutgers and MPH from the University of Medicine and Dentistry's School of Public Health. He has been a software developer for Rutgers for the past six years, initially with the Architecture and Engineering team working on business applications, architecture, and development processes, and now with the Identity Management team, working on open source projects such as Spring Security, CAS and OpenRegistry, as well as participating in the IDM re-architecture initiative at Rutgers. Scott has spoken at various conferences, including Jasig and Educause, on a variety of topics including Spring Security, development best practices, CAS, IDM, and Java development. He has served on the program committee for Internet2's ACAMP and the Jasig Conferences. Scott currently serves as the chairperson of the Jasig CAS Steering Committee, and as the project's lead architect. In his spare time, Scott enjoys running marathons, hiking, photography, learning new libraries and tools, and volunteering for the Make-A-Wish Foundation of New Jersey.

Carlos Sanchez has been involved in open source for nearly ten years. He has specialized in solving business challenges in a wide variety of industries, including e-commerce, financial services, telecommunications, and software development. He is a member of the Apache Maven Project Management Committee (PMC), the Apache software Foundation, the Eclipse Foundation, and a committer in the Spring Security project among others and he is currently the Sr. Solutions Architect at G2iX in Los Angeles.

Carlos completed his Computer Engineering degree at the University of Coruña, Spain, and enjoys traveling and discovering new places.

Carlos has coauthored "Better Builds with Maven"—the first book about Maven 2, and reviewed other books about the project.

Table of Contents

Preface

Welcome to the world of Spring Security 3! I'm certainly pleased that you have acquired the first published book fully devoted to Spring Security, and I hope that it fulfills your every wish for a technical book on this fascinating subject. I'd like to use this introduction to set your expectations for the pages ahead, and give you some advice to help you along your way.

By the time you finish this book, you should feel comfortable with the architecture of Spring Security, the incorporation of Spring Security in a web-based application, and the integration of Spring Security with many types of external authentication and authorization systems.

The book is largely divided into two halves. The first half (Chapters 1-7) covers Spring Security as part of a web application from start to finish—very basic initial setup, all the way to advanced access control list security. The second half (Chapters 8-12) covers Spring Security as part of a larger software ecosystem, illustrating integration with common external systems such as OpenID, Microsoft Active Directory, and LDAP. The final chapter covers migration issues when moving from Spring Security 2 to Spring Security 3.

The book uses a simple Spring Web MVC based application to illustrate the concepts presented in the book. The application is intended to be very simple and straightforward, and purposely contains very little functionality—the goal of this application is to encourage you to focus on the Spring Security concepts, and not get tied up in the complexities of application development. You will have a much easier time following the book if you take the time to review the sample application source code, and try to follow along with the exercises. Some tips on getting started are in *Chapter 1*, *Anatomy of an Unsafe Application* and *Appendix*, *Additional Reference Material*.

What this book covers

Chapter 1, *Anatomy of an Unsafe Application* covers a hypothetical security audit of our e-commerce site, illustrating common issues that can be resolved through proper application of Spring Security. You will learn about some basic security terminology, and review some prerequisites for getting the sample application up and running.

Chapter 2, *Getting Started with Spring Security* reviews basic setup and configuration of form-based authentication with Spring Security, followed by a high-level overview of how Spring Security works from start to finish to secure web requests.

Chapter 3, *Enhancing the User Experience* illustrates additional user-facing functionality supported by Spring Security that can increase the usability of secured sites, including a remember me function, a styled login page, logout, and password change capability.

Chapter 4, *Securing Credential Storage* guides you through the key configuration steps required to secure your users' information in a JDBC database, using Spring Security APIs. Important security concepts around safe password storage are also covered in this chapter.

Chapter 5, *Fine-Grained Access Control* covers in-page authorization checking (partial page rendering), and business-layer security using Spring Security's method security capabilities.

Chapter 6, *Advanced Configuration and Extension* provides several hands-on walkthroughs of common customizations to Spring Security implementations, including custom servlet filters, custom authentication providers, and custom exception handling. Session fixation and concurrent session control are analyzed and appropriately applied to the site with required configuration steps reviewed. Finally, explicit bean-based configuration is clearly illustrated for all Spring Security functionality covered in the book.

Chapter 7, *Access Control Lists* teaches you the concepts and basic implementation of business object-level security using the Spring Security Access Control Lists module—a powerful module with very flexible applicability to challenging business security problems.

Chapter 8, *Opening up to OpenID* covers OpenID-enabled login and user information exchange, as well as a high-level overview of the logical flow of an OpenID-enabled system.

Chapter 9, *LDAP Directory Services* provides a guide to application integration with an LDAP directory server, including practical tips on different types of integrations with LDAP data.

Chapter 10, *Single Sign On with Central Authentication Service* shows how integration with Central Authentication Service (CAS) can provide single sign-on support to your Spring Security-enabled application.

Chapter 11, *Client Certificate Authentication* makes X.509 certificate-based authentication a clear alternative for certain business scenarios where managed certificates can add an additional layer of security to our application.

Chapter 12, *Spring Security Extensions* covers the Spring Security Kerberos extension project, which exposes a Kerberos integration layer for user authentication in our Spring Security application providing compatibility with a Unix Kerberos environment or Microsoft Active Directory.

Chapter 13, *Migration to Spring Security 3* lays out the major differences between Spring Security 2 and Spring Security 3, including notable configuration changes, class and package migrations, and important new features. If you are familiar with Spring Security 2, we recommend that you read this chapter first, as it may make it easier to tie the examples back to code with which you are familiar.

Appendix, *Additional Reference Material* covers some reference material, which we feel is helpful (and largely undocumented) and too comprehensive to insert in the text of the chapters.

Other notes

Please do keep in mind that in the age of open source software, open and rapid development, and the collective wisdom of the Internet, books no longer age gracefully. We'll certainly endeavor to maintain online content related to the book as Spring Security changes, but keep in mind that the examples in this book were written using Spring Security 3.0.x, and may require slight or significant adaptation for the version of the software you are using.

The overall conceptual design of the framework, and certainly discussion of general security topics and integrations are likely to remain fairly stable and accurate for the near future. We're always happy to hear your suggestions and feedback—you may pass along either through the Packt Publishing website, or at the website I personally maintain for the book, `http://www.springsecuritybook.com/`.

This book will:

- Provide you with practical, example-driven implementation advice on the Spring Security Framework.

- Assume that you have some prior knowledge of Java web application development and the concepts therein. Given that the sample application is developed using very basic Spring MVC techniques. We assume you have a limited understanding of how this works (we won't use advanced Spring MVC techniques, so hopefully it is easy to follow even with no prior experience).

- Encourage you to follow along with the sample code and try out what you are reading! While it's not required to have the code up and active in a development environment while you're reading the book, we can guarantee you'll get a lot more out of it if you try to follow along with the examples.

This book will not:

- Teach you every step required to fully secure a production web application. It will teach you the tools that Spring Security provides, and help you learn (and practice) some techniques such as session fixation attacks, but a full survey of security threats is certainly beyond the scope of this book. Where appropriate, we point you at additional reading on the subject of web application security practices. The most important tool you can use in this area is your brain!

- Teach you tangentially related concepts and techniques, such as AOP, Spring MVC, LDAP, CAS, Windows Active Directory, and so on. Although we lightly touch on these topics where relevant in the text, entire books have been written about each of these subjects. Be smart— if you are heavily invested in a project using an advanced technology, take the time to learn it—you'll certainly earn a big return on your time investment.

- Provide you with a complete reference of every attribute, element, and API call. Luke and the Spring Security team and community spend a lot of time keeping the reference manual up to date, and providing very well documented source code. Get familiar with the code—don't be afraid to read the JavaDoc—and use the examples and diagrams in this book to wire it all together mentally. We have tried to cover the most common Spring Security usage scenarios from top to the very deep bottom. This approach was taken because we felt that providing the user with a complete understanding of the most common topics would help them grasp additional topics with much less effort. We apologize in advance if we do not cover your favorite feature of Spring Security in detail!

Some notation conventions we've tried to use in the book to help you out:

- XML elements will always be surrounded with angle brackets, like `<intercept-url>`, to differentiate them from XML attributes, which will always be in code font, like `use-expressions`.

- For conciseness, we've abbreviated the base package name `org.springframework.security` as `o.s.s` in fully-qualified class names found in the text and diagrams. We retain the full package names in code and configuration samples, to avoid confusion. We try to list the fully-qualified class names upon first mention of a particular class, so that curious folks can know exactly where to look for a class without hunting.

- Also for conciseness, we've generally omitted import statements from Java code listings, and in some cases replaced unimportant, repeated configuration elements with an XML comment `<!-- ... -->`—this is used to focus your attention on the bits of code or configuration that we're describing in a particular example, and shouldn't be taken as an indication that there shouldn't be anything there!

- Due to the length of some fully-qualified class names or XML configuration lines, the code listings may wrap. We assume you know enough about Java and XML to recognize where a line break makes sense and where it doesn't. Remember that the source code for the book contains all the examples in case you're not sure!

Acknowledgements and thanks

Some acknowledgements—writing a technical book involves an unbelievable amount of research, review, support, and most of all—time, especially while holding down a demanding full-time job and being a devoted dad! I vow never again to skim text in a technical book <grin>.

Thank you to the Packt Publishing crew for helping this first-time author through the very long process of publishing a technical book: Steven Wilding for approaching me with the idea for the book, Poorvi Nair for keeping me on track, Neha Patwari and Meeta Rajani for final review and edits, and Patricia Weir for all the overall help. To Luke Taylor, Ben Alex, and the other contributors to the Spring Security/ Acegi project, thank you for building a wonderfully complex, interesting, and above all useful framework. To Scott Battaglia and Carlos Sanchez, thank you for your dedication during the technical review process—your feedback was exceedingly valuable and much appreciated.

To Mark Pitts, my friend and long-time co-worker, thanks for the encouragement along the way—you will get a signed copy! To Ganesh Murthy, Matt Olson, and Kris Huggins, thanks for the encouragement! To Jon Burns, Thierry Hubert, and Matt Crowe, you guys are entrepreneurial inspirations. To the Voodoo Lounge crew, long live the Voodoo Lounge! Thanks as well to all the aTao folks, old and new.

Thanks also to all the contributors to the Spring Security support forum— answering your questions has continually improved my understanding of Spring Security, and ultimately resulted in a much richer and more complete book.

Finally, to my wife and kids (once again!) for, most of all, the tolerance and support for my part-time hobby—unfortunately, it won't make a very exciting bedtime story!

Who this book is for

This book is for Java developers who build web projects and applications. The book assumes basic familiarity with Java, XML, and the Spring Framework. Newcomers to Spring Security will still be able to utilize all aspects of this book.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code will be set as follows:

```
<c:url value="/j_spring_security_logout" var="logoutUrl"/>
<li><a href="${logoutUrl}">Log Out</a></li>
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items will be shown in bold:

```
<http auto-config="true" use-expressions="true">
  <logout invalidate-session="true"
      logout-success-url="/"
      logout-url="/logout"/>
</http>
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in our text like this: "Once you complete these changes, restart the application and look for the **Change Password** action under the **My Account** section of the site."

> Warnings or important notes appear in a box like this.

> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply drop an email to `feedback@packtpub.com`, and mention the book title in the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or email `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

> **Downloading the example code for the book**
> Visit `https://www.packtpub.com//sites/default/files/downloads/9744_Code.zip` to directly download the example code.
> The downloadable files contain instructions on how to use them.

Errata

Although we have taken every care to ensure the accuracy of our contents, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in text or code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration, and help us to improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **let us know** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata added to any list of existing errata. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

1
Anatomy of an Unsafe Application

Security is arguably one of the most critical architectural components of any web-based application written in the 21st century. In an era where malware, criminals, and rogue employees are always present and actively testing software for exploits, smart and comprehensive use of security is a key element to any project for which you'll be responsible.

This book is written to follow a pattern of development that we feel provides a useful premise for tackling a complex subject—taking a web-based application with a Spring 3 foundation, and understanding the core concepts and strategies for securing it with Spring Security 3.

Whether you're already using Spring Security or are interested in taking your basic use of the software to the next level of complexity, you'll find something to help you in this book.

During the course of this chapter, we'll:

- Review the results of a fictional security audit
- Discuss some common security problems of web-based applications
- Learn several core software security terms and concepts

If you are already familiar with basic security terminology, you may directly skip to *Chapter 2, Getting Started with Spring Security*, where we dig into the details of the framework.

Security audit

It's early in the morning at your job as software developer for the Jim Bob Circle Pants Online Pet Store (`JBCPPets.com`), and you're halfway through your first cup of coffee when you get the following e-mail from your supervisor:

To: Star Developer <stardev@jbcppets.com>

From: Super Visor <theboss@jbcppets.com>

Subject: Security Audit

Star,

We have a third-party security company auditing our e-commerce storefront today. Please be available to fix any issues they might uncover, although I know you took security into account when you designed the site.

Super Visor

What? You didn't think a lot about security when you designed the application? Sounds like you'll have a lot to learn from the security auditors! First, let's spend a bit of time examining the application that's under review.

About the sample application

Although we'll be working through a contrived scenario as we progress through this book, the design of the application and the changes we'll make to it are drawn from real-world usage of Spring-based applications.

The application is designed to be simplistic, to allow us to focus on the important aspects of security and not get tied up in the details of ORM and complex UI techniques. We expect you to refer to other supplementary material to cover some of the baseline functionality that is provided as part of the sample code.

The code is written in Spring and Spring Security 3, but it would be relatively easy to adapt many of the examples to Spring Security 2. Refer to the discussion about the detailed changes between Spring Security 2 and 3 in *Chapter 13, Migration to Spring Security 3,* for assistance in translating the examples to Spring Security 2 syntax.

Please don't use this application as a baseline to build a real online pet store. It has been purposely structured to be simple and to focus on the concepts and configuration that we illustrate in the book.

The JBCP pets application architecture

The web application follows a standard three-tier architecture, consisting of web, service, and data access layers, as indicated in the following diagram:
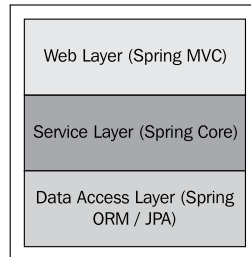
The **web layer** encapsulates MVC code and functionality. In this sample application, we use the Spring MVC framework, but we could just as easily use Spring Web Flow, Struts, or even a Spring-friendly web stack such as Apache Wicket.

In a typical web application leveraging Spring Security, the web layer is where much of the configuration and augmentation of code takes place. As such, if you haven't had a lot of experience with web applications, and Spring MVC specifically, it would be wise to review the baseline code closely and make sure you understand it before we move on to more complex subjects. Again, we've tried to make the website as simple as possible, and the construct of an online pet store is used just to give a sensible title and light structure to the site. This approach is in contrast to the complex Java EE Pet Clinic sample on which many tutorials are based.

The **service layer** encapsulates the business logic for the application. In our sample application, we make this a very light facade over the data access layer, to illustrate particular points around securing application service methods.

In a typical web application, this layer would incorporate business rules validation, composition and decomposition of business objects, and cross-cutting concerns such as auditing.

The **data access layer** encapsulates the code responsible for manipulating contents of database tables. In many Spring applications, this is where you would see the use of an ORM such as Hibernate or JPA. It exposes an object-based API to the service layer. In our sample application, we use basic JDBC functionality to achieve persistence to the in-memory HSQL database.

In a typical web application, a more comprehensive data access solution would be utilized. As ORM, and specifically data access, tend to be confusing for some developers, this is an area we have chosen to simplify, as much as possible, for the purposes of clarity.

Application technology

We have endeavored to make the application as easy to run as possible by focusing on some basic tools and technologies that almost every Spring developer would have on their development machine. Nevertheless, we provide the supplementary Getting Started information in *Appendix*, *Additional Reference Material*.

We recommend the following IDEs for improved productivity when working with Spring Security, with the sample code in the book:

- Eclipse 3.4 or 3.5 Java EE Bundle available at `http://www.eclipse.org/downloads/`

- Spring IDE 2.2 (2.2.2 or higher) available at `http://springide.org/blog/`

In some of the examples and screenshots in this book, you'll see Eclipse and the Spring IDE tools used, so we'd really recommend that you give them a try.

You may also wish to try the free **Spring Tool Suite** (**STS**) distribution of Eclipse—a bundle of Eclipse distributed by Spring Source that includes the next generation of Spring IDE functionality (available at `http://www.springsource.com/products/springsource-tool-suite-download`). Some users dislike the inclusiveness and Spring Source branding of this IDE, but if you are doing significant Spring development, it has a number of helpful features.

Primarily, we provide Eclipse 3.4-compatible projects that allow you to build the code within Eclipse, and deploy instructions to Tomcat 6.x server. As many developers are comfortable with Eclipse, we felt this was the most straightforward method of packaging the examples. We have also provided Apache Ant build scripts for these samples, as well as Apache Maven modules. Whatever development environment you are familiar with, hopefully you will find a way to work through the examples while you read the book.

Also, you'll want to download the full releases of both Spring 3 and Spring Security 3. The Javadoc and source code are top notch if you get confused or want more information, and the samples can provide an additional level of support or reassurance in your learning.

Reviewing the audit results

Let's return to our e-mail and see how the audit is progressing. Uh-oh, the results don't look good:

To: Star Developer <stardev@jbcppets.com>

From: Super Visor <theboss@jbcppets.com>

Subject: FW: Security Audit Results

Star,

Have a look at the results and come up with a plan to address these issues.

Super Visor

APPLICATION AUDIT RESULTS

This application exhibits the following insecure behavior:

- Inadvertent privilege escalation due to lack of URL protection and general authentication
- Inappropriate or non-existent use of authorization
- Database credentials not secured and easily accessible
- Personally identifiable or sensitive information is easily accessible or unencrypted
- Insecure transport-level protection due to lack of SSL encryption
- Risk level: HIGH

We recommend that this application be taken offline until these issues can be resolved.

Ouch! This result looks bad for our company. We'd better work to resolve these issues as quickly as possible.

Third-party security specialists are often hired by companies (or their partners or customers) to audit the effectiveness of their software security, through a combination of white hat hacking, source code review, and formal or informal conversations with application developers and architects.
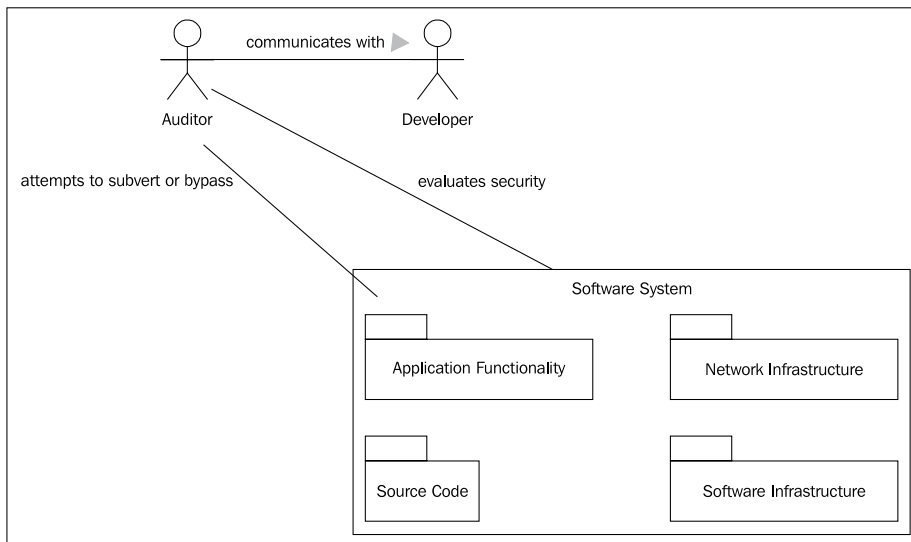
The goal of security audits is typically to provide management or clients with an assurance that basic secure development practices have been followed to ensure integrity and safety of customer data and system function. Depending on the industry the software is targeted for, the auditor may also test using industry-specific standards or compliance metrics.

Two specific security standards that you're likely to run into at some point in your career are the **Payment Card Industry Data Security Standard (PCI DSS)** and the **Health Insurance Privacy and Accountability Act (HIPAA)** privacy rules. Both standards are intended to ensure safety of specific sensitive information (credit card and medical information respectively) through a combination of process and software controls. Many other industries and countries have similar rules around sensitive or **Personally Identifiable Information (PII)**. Failure to follow these standards is not only a bad practice, but something that could expose you or your company to significant liability (not to mention bad press!) in the event of a security breach.

Receiving the results of a security audit can be an eye-opening experience; however, following through with the required software improvements can be a perfect opportunity for self-education and software improvement, and can allow you to implement practices and policies that lead to secure software.

Let's review the auditor's findings and come up with a plan to address them in detail.

Authentication

Inadvertent privilege escalation due to lack of URL protection and general authentication.

Authentication is one of two key security concepts that you must internalize when developing secure applications (the other being authorization). Authentication deals with the specific identification of a single user of the system, and the mapping of that user to an identifiable (and therefore securable) entity. Typically, a software system will be divided into two high-level realms such as unauthenticated (or **anonymous**) and authenticated as shown in the following figure:
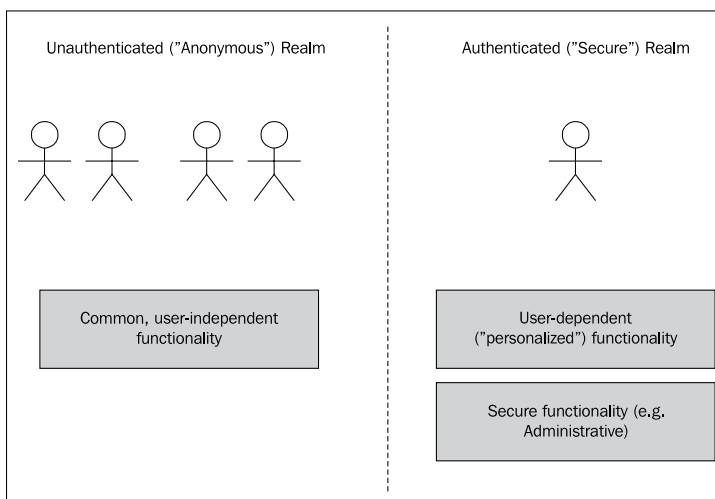
Application functionality in the anonymous realm is the functionality that is independent of a user's identity (think of a product listing from an online store).

Anonymous areas do not:

- Require a user to log into the system or otherwise identify themselves to be usable
- Display sensitive information such as names, addresses, credit cards, orders, and so on
- Provide functionality to manipulate the overall state of the system or its data

Unauthenticated areas of the system are intended for use by everyone, even by users who we haven't specifically identified yet. However, it may be that, additional functionality appears to identified users in these areas (for example, the ubiquitous **Welcome {First Name}** text). Selective display of content to authenticated users is fully supported through use of the Spring Security tag library, and is covered in *Chapter 5, Fine-Grained Access Control*.

We'll resolve this finding and implement form-based authentication using Spring Security's automatic configuration capability in Chapter 2. More complex forms of authentication (which usually revolve around systems integration with enterprise or other external authentication stores) are covered in the second half of this book, starting with *Chapter 8, Opening up to OpenID*.

Authorization

Inappropriate or non-existent use of authorization.

Authorization is the second of two core security concepts that is crucial in implementing and understanding application security. **Authorization** deals with the appropriate availability of functionality and data to users who are authorized (allowed) to access it. Built around the authorization model for the application is the activity of partitioning the application functionality and data such that availability of these items can be controlled by matching the combination of privileges, functionality and data, and users. Our application's failure on this point of the audit indicates that the application functionality isn't restricted by user role. Imagine if you were running an e-commerce site and the ability to view, cancel, or modify order and customer information was available to any user of the site!

We'll address the basic authorization problem with Spring Security's authorization infrastructure in Chapter 2, followed by more advanced authorization, both at the web tier in Chapter 5, and the business tier in *Chapter 6*, *Advanced Configuration and Extension*.

Database Credential Security

Database credentials not secured and easily accessible.

Through their examination of the application source code and configuration files, the auditors noted that user passwords were stored in plain text in the configuration files, making it very easy for a malicious user with access to the server to gain access to the application.

As the application contains personal and financial data, a rogue user being able to access any data could expose the company to identity theft or tampering. Protecting access to the credentials used to access the application should be a top priority for us, and an important first step is ensuring one point of failure in security does not compromise the entire system.

We'll examine the configuration of Spring Security's database access layer for credential storage, which uses JDBC connectivity, in *Chapter 4*, *Securing Credential Storage*. In Chapter 4, we'll also look at built-in techniques to increase the security of passwords stored in the database.

Sensitive Information

Personally identifiable or sensitive information is easily accessible or unencrypted.

The auditors noted that some significant and sensitive pieces of data, including credit card numbers, were completely unencrypted or masked anywhere in the system. Aside from presenting a clear case of poor data design, lack of protection around credit card numbers is a violation of the PCI standard.

Fortunately, there are some simple design patterns and tools that allow us to protect this information securely with Spring Security's annotation-based AOP support. We'll examine several techniques for securing this type of data at the data access layer in Chapter 5.

Transport-Level Protection

Insecure transport-level protection due to lack of SSL encryption.

While in the real world, it's unthinkable that an online commerce website would operate without SSL protection; unfortunately JBCP Pets is in just this situation. SSL protection ensures that communication between the browser client and the web application server are secure against many kinds of tampering and snooping.

In Chapter 5, we'll review the basic options for using transport-level security as part of the definition of the secured structure of the application, including planning which areas of the application will be covered by forced SSL encryption.

Using Spring Security 3 to address security concerns

Spring Security 3 provides a wealth of resources that allow for many common security practices to be declared or configured in a straightforward manner. In the coming chapters, we'll apply a combination of source code and application configuration changes to address all of the concerns raised by the security auditors (and more), and give ourselves confidence that our e-commerce application is secure.

With Spring Security 3, we'll be able to make the following changes to increase our application security:

- Segment users of the system into user classes
- Assign levels of authorization to user roles
- Assign user roles to user classes
- Apply authentication rules globally across application resources
- Apply authorization rules at all levels of the application architecture
- Prevent common types of attacks intended to manipulate or steal a user's session

Why Spring Security?

Spring Security exists to fill a gap in the universe of Java third-party libraries, much as the Spring Framework originally did when it was first introduced. Standards such as Java Authentication and Authorization Service (JAAS) or Java EE Security do offer some ways of performing some of the same authentication and authorization functions, but Spring Security is a winner because it packages up everything you need to implement a top-to-bottom application security solution in a concise and sensible way.

Additionally, Spring Security appeals to many because it offers out of the box integration with many common enterprise authentication systems; so it's adaptable to most situations with little effort (beyond configuration) on the part of the developer.

It's in wide use because there's really no other mainstream framework quite like it!

Summary

In this chapter, we have:

- Reviewed common points of risk in an unsecured web application
- Reviewed the basic architecture of the sample e-commerce application
- Discussed the strategies for securing the e-commerce application

In Chapter 2, we'll examine Spring Security's overall architecture, with a specific eye towards the many places where it can be extended and configured to support your application's needs.

2
Getting Started with Spring Security

In this chapter, we'll review the core concepts behind Spring Security, including important terminology and product architecture. We'll look at some of the options for configuring Spring Security, and what effect they have on the application.

Most importantly, to save our jobs, we'll have to start securing the JBCP Pets online store! We'll address our first finding—*inadvertent privilege escalation due to lack of URL protection and general authentication*—from the security audit discussed in *Chapter 1, Anatomy of an Unsafe Application* by analyzing and understanding how to ensure that authentication exists to protect appropriate areas of the store.

During the course of this chapter, we'll:

- Become aware of the key applications of security concepts
- Implement a basic level of security on the JBCP Pets online store, using Spring Security's quick setup option
- Understand the high level architecture of Spring Security
- Explore standard configurations and options for authentication and authorization
- Examine the use of Spring Expression Language in Spring Security access control

Core security concepts

Since the results of that enlightening security audit, you've done some research and determined that Spring Security provides a solid foundation on which you can build a system to address the security issues identified in the Spring Web MVC-based JBCP Pets online store.

In order to make effective use of the Spring Security product, it's important to review some key concepts and terminology before we start assessing and improving our application's security profile.

Authentication

As we discussed in Chapter 1, authentication is the process of verifying that the users of our application are who they say they are. You may be familiar with authentication in your daily online and offline life, in very different contexts:

- **Credential-based authentication**: When you log in to your web-based e-mail account, you most likely provide your username and password. The e-mail provider matches your username with a known user in its database, and verifies that your password matches with what they have on record. These credentials are what the e-mail system uses to validate that you are a valid user of the system. First, we'll use this type of authentication to secure sensitive areas of JBCP Pets online store. Technically speaking, the e-mail system can check credentials not only in the database but anywhere—for example, a corporate directory server such as Microsoft Active Directory. Some of these types of integrations are covered in the second half of this book.

- **Two-factor authentication:** When you withdraw money from your bank's automated teller machine, you swipe your ID card and enter your personal identification number before you are allowed to retrieve cash or conduct other transactions. This type of authentication is similar to username and password authentication, except that the username is encoded on the card's magnetic strip. The combination of the physical card and user-entered PIN allows the bank to ensure that you should have access to the account. The combination of a password and a physical device (your plastic ATM card) is a ubiquitous form of two-factor authentication. In a professional, security conscious environment, it's common to see these types of devices in regular use for access to highly secure systems, especially dealing with finance or personally identifiable information. A hardware device such as RSA's SecurID combines a time-based hardware device with server-based authentication software, making the environment extremely difficult to compromise.

- **Hardware authentication**: When you start your car in the morning, you slip your metal key into the ignition and turn it to get the car started. Although it may not feel similar to the other two examples, the correct match of the bumps on the key and the tumblers in the ignition switch function as a form of hardware authentication.

There are literally dozens of forms of authentication that can be applied to the problem of software and hardware security, each with their own pros and cons. We'll review some of these methods as they apply to Spring Security in later part of this book; in fact, the entire second half of this book is devoted to many of the most common methods of authentication implementation using Spring Security.

Spring Security extends the Java standard security concept of an authenticated **principal** (`java.security.Principal`), which is used to uniquely represent any one authenticated entity. Although a typical principal is a one-to-one mapping to a single user of the system, it could also correspond to any client of the system, such as a web service client, an automated batch feed, and so on. In most cases, with your use of Spring Security, a `Principal` will simply represent a single user, so when we refer to a principal, you can usually equate this with "user".

Authorization

Authorization typically involves two separate aspects that combine to describe the accessibility of the secured system.

The first is the mapping of an authenticated principal to one or more **authorities** (often called **roles**). For example, a casual user of your website might be viewed as having visitor authority while a site administrator might be assigned administrative authority.

The second is the assignment of authority checks to **secured resources** of the system. This is typically done at the time a system is developed, either through explicit declaration in code or through configuration parameters. For example, the screen that manages our pet store's inventory should be made available only to those users having administrative authority.

> A secured resource may be any aspect of the system that should be conditionally available based on the authority of the user.
>
> Secured resources of a web-based application could be individual web pages, entire portions of the website, or portions of individual pages. Conversely, secured business resources might be method calls on classes or individual business objects.

You might imagine an authority check that would examine the principal, look up its user account, and determine if the principal is in fact an administrator. If this authority check determines the principal attempting to access the secured area is, in fact, an administrator, then the request will succeed. If, however, the principal does not have sufficient authority, the request should be denied.

Let's take a closer look at the example of a particular protected resource, the **Inventory: Edit** page. The **Inventory Edit** page requires administrative access (after all, we don't want regular users adjusting our inventory levels!), and as such looks for a certain level of authority in the principal accessing it.

If we think about how a decision might be made when a site administrator attempts to access the protected resource, we'd imagine that the examination of actual authority versus required authority might be expressed concisely in terms of set theory. We might then choose to represent this decision as a Venn diagram for the administrative user:
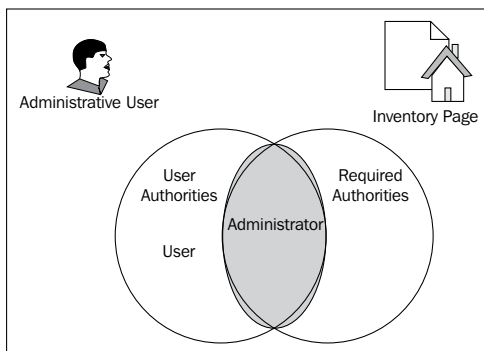
There is an **intersection** between the **User Authorities** (**User** and **Administrator**) and the **Required Authorities** (**Administrator**) for the page, so the user is provided with access.

Contrast this with an unauthorized visitor:
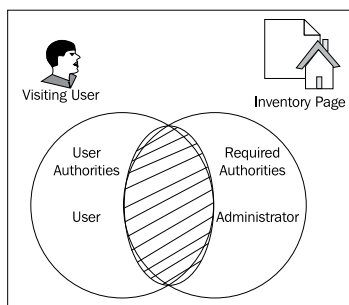
The sets of authorities are **disjoint**, and have no common elements. So, the user is denied access to the page. Thus, we have demonstrated the basic principle of authorization of access to resources.

In reality, there's real code making this decision with consequences of the user being granted or denied access to the requested protected resource. The following diagram illustrates this high-level process, as it applies to Spring Security:
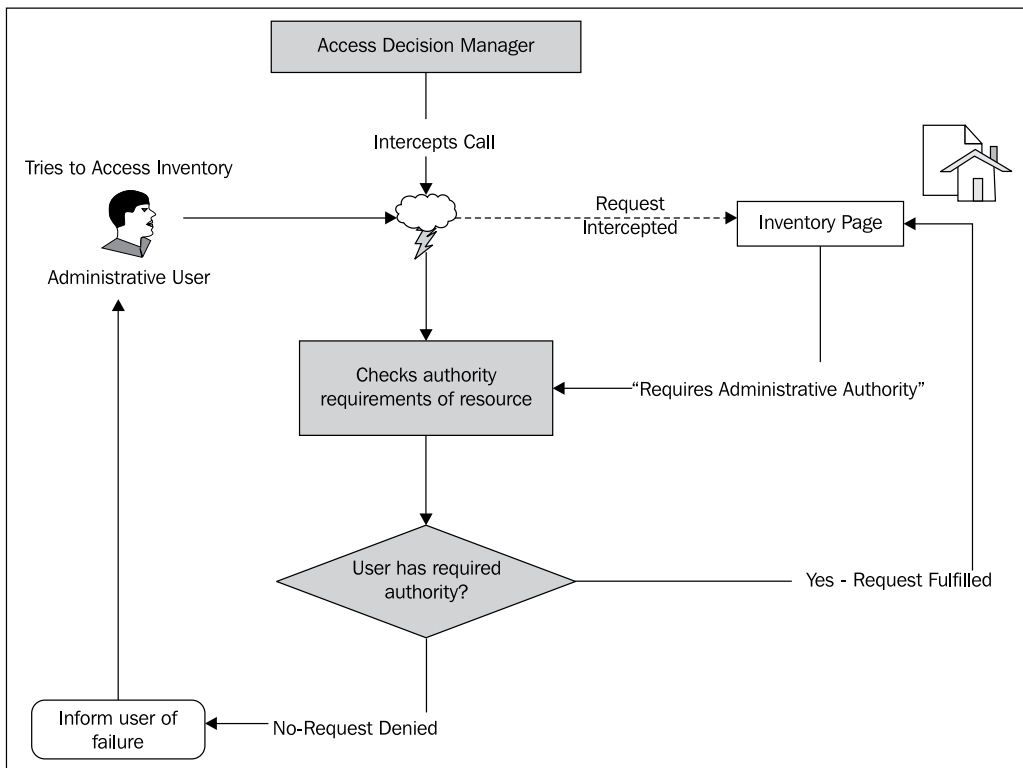
We can see that a component called the **access decision manager** is responsible for determining whether a principal has the appropriate level of access, based on the match between the authority possessed by the principal and the authority requested by the resource.

The access decision manager's evaluation of whether or not access is allowed can be as simple as evaluating the intersection of the set of authorities belonging to the principal, and the authorities required for access to the protected resource.

Let's get to work putting a basic implementation of Spring Security in place for JBCP Pets, and then we'll revisit authentication and authorization in much more detail.

Securing our application in three easy steps

Although Spring Security can be extremely difficult to configure, the authors of the product have been thoughtful and have provided us with a very simple mechanism to enable much of the software's functionality with a strong baseline. From this baseline, additional configuration will allow a fine level of detailed control over the security behavior of our application.

We'll start with our unsecured online store and, with three steps, turn it into a site that's secured with a rudimentary username and password authentication. This authentication serves merely to illustrate the steps involved in enabling Spring Security for our web application; you'll see that there are some obvious flaws in this approach that will lead us to make further configuration refinements.

Implementing a Spring Security XML configuration file

The first step in the baseline configuration process is to create an XML configuration file, representing all Spring Security components required to cover standard web requests.

Create an XML file in the `WEB-INF` directory with the name `dogstore-security.xml`. The file should have the following contents:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans:beans xmlns="http://www.springframework.org/schema/security"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:beans="http://www.springframework.org/schema/beans"
        xsi:schemaLocation="
        http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/
                spring-security-3.0.xsd">
  <http auto-config="true">
    <intercept-url pattern="/*" access="ROLE_USER"/>
  </http>
  <authentication-manager alias="authenticationManager">
    <authentication-provider>
      <user-service>
        <user authorities="ROLE_USER" name="guest" password="guest"/>
```

```
        </user-service>
      </authentication-provider>
    </authentication-manager>
  </beans:beans>
```

This is the only Spring Security configuration required to get our web application secured with a minimal standard configuration. This style of configuration, using a Spring Security-specific XML dialect is known as the `security` namespace style, named after the XML namespace (`http://www.springframework.org/schema/security`) associated with the XML configuration elements. We will discuss an alternate configuration style, using traditional Spring Bean wiring as well as XML namespaces, in *Chapter 6*, *Advanced Configuration and Extension*.

> Users who dislike Spring's XML configuration will be disappointed to learn that there isn't an alternative annotation-based configuration mechanism for Spring Security, as there is with the Spring Framework.
>
> This makes sense if you think through it, as Spring Security is mostly concerned about the behavior of an entire system rather than individual objects and classes. Still, perhaps in the future, we'll see the ability to annotate Spring MVC controllers with security declarations instead of specifying URL patterns in a configuration file!

Although annotations are not prevalent in Spring Security, certain aspects of Spring Security that apply security elements to classes or methods are, as you'd expect, available via annotations. We'll cover these in *Chapter 5*, *Fine-Grained Access Control*.

Adding the Spring DelegatingFilterProxy to your web.xml file

Spring Security's primary influence over our application is through a series of `ServletRequest` filters (we'll explain this when we review the Spring Security architecture later in this Chapter). Think of these filters as a sandwich of security functionality that's layered around every request to our application.

The `o.s.web.filter.DelegatingFilterProxy` is a servlet filter that allows Spring Security to wrap all application requests and ensure that they are appropriately secured.

> The `DelegatingFilterProxy` is actually supplied by the Spring framework, and isn't security specific. This filter is commonly used with Spring-based web applications to bind Spring Bean dependencies to servlet filters in conjunction with the servlet filter lifecycle.

Configure a reference to this filter by adding the following code to our `web.xml` deployment descriptor, right after the end of our Spring MVC `<servlet-mapping>` element:

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filterclass>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

What we're doing is applying a `ServletRequest` filter and configuring it to handle requests matching the given URL pattern (`/*`). As the wildcard pattern we've provided matches all URLs, this filter will be applied to every request.

If you're paying attention, you'll note that this doesn't have anything to do with the Spring Security configuration that we performed in `dogstore-security.xml` yet. For that, we'll have to add an XML configuration file reference to our web application deployment descriptor, `web.xml`.

Adding the Spring Security XML configuration file reference to web.xml

Depending on how you have configured your Spring web application, you may or may not have an explicit reference to an XML configuration file in your `web.xml` deployment descriptor. The default behavior of the Spring Web `ContextLoaderListener` is to look for an XML configuration file of the same name as your Spring Web servlet. Let's walk through the addition of the new Spring Security XML configuration file to the existing JBCP Pet Store baseline site.

First, we have to carefully see if the site is using the Spring MVC automatic lookup for an XML configuration file. To do this, we look at the `web.xml` file to see the name of the servlet, indicated in the `<servlet-name>` element:

```
<servlet>
  <servlet-name>dogstore</servlet-name>
  <servletclass>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

As the name of the servlet (`<servlet-name>`) is `dogstore`, Spring's **Convention over Configuration** (**CoC**) rules will search for an XML configuration file called `dogstore-servlet.xml` in `WEB-INF`. We haven't overridden this default behavior, and you can find this file, with very little Spring MVC-related configuration, in the `WEB-INF` directory.

> Many users of Spring Web Flow or Spring MVC really don't understand how these CoC rules are formulated, or where in the Spring code to get clarity of these rules. `o.s.web.context.support.XmlWebApplicationContext` and its superclasses are good places to start. The Javadoc does a good job of explaining the configuration parameters specific to web applications based on the Spring Web MVC framework.

It is also possible to declare additional spring `ApplicationContext` configuration files that will be loaded prior to the configuration file related to the Spring MVC servlet. This is done using the Spring `o.s.web.context.ContextLoaderListener` to create an `ApplicationContext` independent of the Spring MVC `ApplicationContext`. This is typically how non-Spring MVC beans are configured, and is usually a good place to add Spring security configuration as well.

The location of XML files used to configure the `ContextLoaderListener` is listed in a `<context-param>` element of our web application deployment descriptor:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/dogstore-base.xml
  </param-value>
</context-param>
```

Remember that the `dogstore-base.xml` file would typically contain standard Spring bean definitions such as data sources, service beans, and so on. We can now add a reference to a new XML configuration file to store Spring Security configuration information, as seen in the updated `<context-param>` element:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    /WEB-INF/dogstore-security.xml
    /WEB-INF/dogstore-base.xml
  </param-value>
</context-param>
```

After we put the new Spring Security configuration file in the web deployment descriptor and restart our web application, try hitting the home page at `http://localhost:8080/JBCPPets/home.do`, and you'll be presented with the following screen:
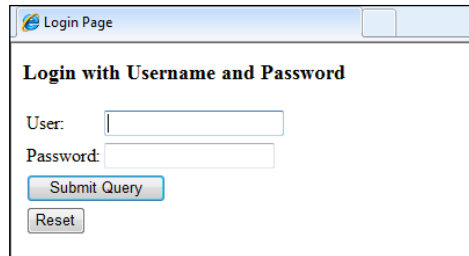
Great job! We've implemented a basic layer of security in our application using Spring Security in just three steps. At this point, you should be able to log in using **guest** as the username and password. You'll see the very basic JBCP Pets home page.

Note that for simplicity, the source code for this chapter contains a very small subset (a single page) of the complete JBCP Pets application. This is done for the sake of simplicity and allows you to focus on getting the application built and deployed without worrying about the additional functionality that will be demonstrated in later chapters. It also provides a great baseline for you to experiment with the configuration parameters and redeploy the application very quickly to test them out!

Mind the gaps!

Stop at this point and think about what we've just built. You may have noticed some obvious issues that will require some additional work and knowledge of the Spring Security product before we are production ready. Try to make a list of the changes in your mind that you think are required before this security implementation is ready to roll out to the public-facing website.

Setting up the baseline profile of our Spring Security implementation was blindingly fast, and has provided us with a login page, username and password authentication, as well as automatic interception of URLs in our online store. However, there are gaps between what the automatic configuration setup provides and what our end goal is:

- We've had to hardcode the username, password, and role information of the user in the XML configuration file. Do you remember this section of XML we added:

```
<authentication-manager alias="authenticationManager">
        <authentication-provider>
```

```
            <user-service>
                   <user authorities="ROLE_USER" name="guest"
password="guest"/>
               </user-service>
       </authentication-provider>
</authentication-manager>
```

You can see that the username and password are right there in the file. It would be unlikely that we'd want to add a new XML declaration to the file for every user of the system! To address this, we'll need to replace this with a database-based authentication provider (we will do this in *Chapter 4, Securing Credential Storage*).

- We've locked down all pages in the store, including pages that a potential customer would want to browse anonymously. We'll need to refine the roles required to accommodate anonymous, authenticated, and administrative users (this is also covered in Chapter 4).

- While the login page is helpful, it's completely generic and doesn't look like our JBCP Store at all. We should add a login form that's integrated with our store's look and feel (we'll tackle this in the next chapter).

Common problems

Many users have trouble with the initial implementation of Spring Security in their application. A few common issues and suggestions are listed next. We want to ensure you can run the example application and follow along!

- Make sure you can build and deploy the application before putting Spring Security in place. Review some introductory samples and documentation on your servlet container if needed.

- It's usually easiest to use an IDE such as Eclipse to run your servlet container. Not only deployment is typically seamless, but the console log is also readily available to review for errors. You can also set breakpoints at strategic locations, which are triggered on exceptions to better diagnose errors.

- If your XML configuration file is incorrect, you will get this (or something similar to this): **org.xml.sax.SAXParseException: cvc-elt.1: Cannot find the declaration of element 'beans'.** It's quite common for users to get confused with the various XML namespace references required to properly configure Spring Security. Review the samples again, paying attention to be careful of line wrapping in the schema declarations, and use an XML validator to verify that you don't have any malformed XML.

- Make sure the versions of Spring and Spring Security that you're using match, and that there aren't any unexpected Spring JARs remaining as part of your application.

Security is complicated: The architecture of secured web requests

The three step configuration that we illustrated before was impressively quick to implement; thanks to the Spring Security's powerful baseline configuration features and sensible out of the box defaults for authentication, which we enabled through the presence of the `auto-config` attribute on the `<http>` element.

Unfortunately, application implementation concerns, architecture limitations, and infrastructure integration requirements are likely to complicate your implementation of Spring Security well beyond what this simple configuration provides. Many users of Spring Security run into trouble moving beyond the basic configuration, as they don't understand the architecture of the product and how all the factors work together to define a cohesive whole.

Comprehending the overall flow of web requests and how they move through the chain of responsibility is crucial to our success with advanced topics in Spring Security. Keep in mind the basic concepts of authentication and authorization as they fit into the overall architecture of our protected system.
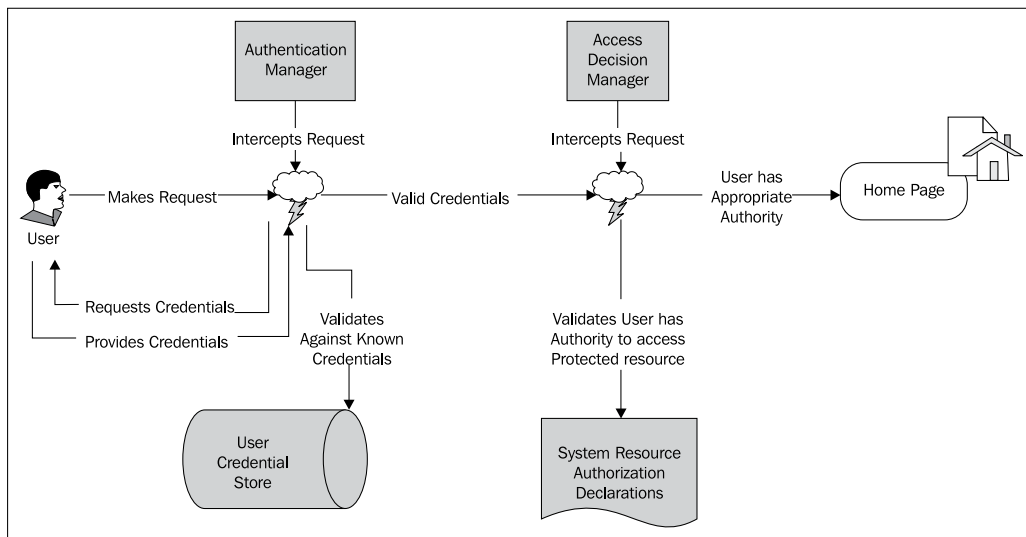
How requests are processed?

The Spring Security architecture relies heavily on the use of **delegates** and **servlet filters** to provide layers of functionality around the context of a web application request.

Servlet Filters (classes that implement the `javax.servlet.Filter` interface) are used to intercept user requests and perform pre-or post-processing, or redirect the request altogether, depending on the function of the servlet filter. The final destination servlet is the Spring MVC dispatcher servlet, in the case of the JBCP Pets online store, but in theory, it could represent any web servlet. The following diagram illustrates how a servlet filter wraps a user's web request:
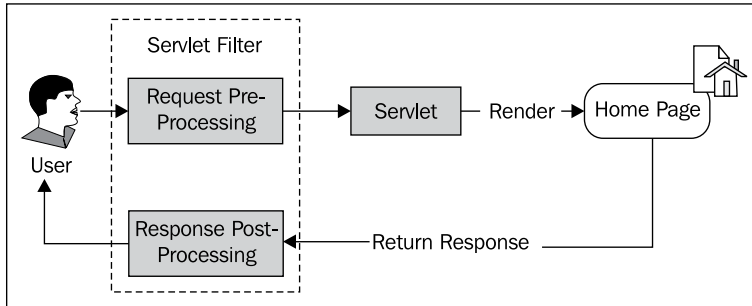
The automatic configuration attribute in the Spring Security XML configuration file sets up a series of ten servlet filters, which are applied in a sequence through the use of a Java EE servlet filter chain. The **filter chain** is a Java EE Servlet API concept specified by the `javax.servlet.FilterChain` interface that allows a web application to direct that a chain of servlet filters should apply to any given request.

Similar to a physical chain made from metal links, each servlet filter represents a link in the chain of method calls used to process the user's request. Requests travel along the chain, being processed by each filter in turn.
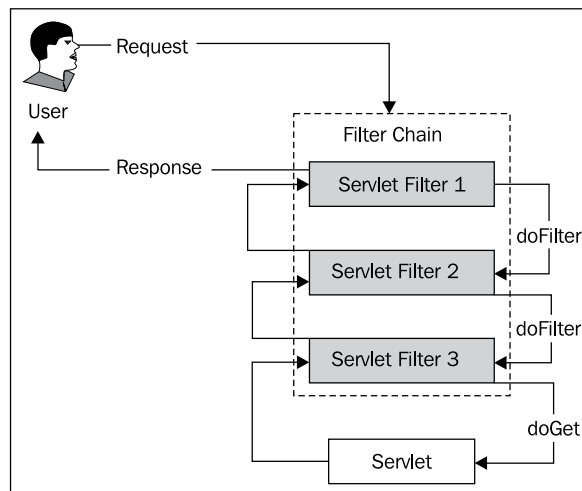
As you can infer from the analogy of a chain, servlet requests moving along a filter chain pass from one filter to the next, in a well-defined order, finally arriving at the destination servlet. Conversely, after the servlet handles the request and returns a response, the filter chain call stack unwinds back through all the filters.

Spring Security takes the filter chain concept and implements its own abstraction, the `VirtualFilterChain`, which provides the added functionality of allowing a `FilterChain` to be dynamically constructed based on URL patterns as specified in the Spring Security XML configuration (contrast this to the standard Java EE `FilterChain`, which is specified based on the web application's deployment descriptor).

> Servlet filters can be used for many purposes besides the filtering (or request blocking) that their name might imply. In fact, many servlet filters function as a proxy for AOP-style interception of web requests in a web application runtime environment, as they allow functionality to occur before and after any web request to the servlet container.
>
> This multipurpose nature of filters is true in Spring Security as well, as many of the filters do not directly impact the user's request.

The automatic configuration option sets up 10 Spring Security filters for you. Understanding what these default filters do, and where and how they are configured, is critical to advanced work with Spring Security.

These filters, and the order in which they are applied, are described in the following table. Most of these filters will be described again as we proceed through our work on the JBCP Pets online store, so don't worry if you don't understand exactly what they do now.

Filter name	Description
`o.s.s.web.context.` `SecurityContextPersistenceFilter`	It is responsible for loading and storage of the `SecurityContext` from the `SecurityContextRepository`. The `SecurityContext` represents the user's secured, authenticated session.
`o.s.s.web.authentication.logout.` `LogoutFilter`	It watches for a request to the virtual URL specified as the logout URL (by default, `/j_spring_security_logout`), and logs the user out if there is a match.

Filter name	Description

`o.s.s.web.authentication.`
`UsernamePasswordAuthentication`
`Filter`
	It watches for a request to the virtual URL used for form-based authentication with username and password (by default, `/j_spring_security_check`), and attempts to authenticate the user if there is a match.

`o.s.s.web.authentication.ui.` `DefaultLoginPageGeneratingFilter`	It watches for a request to the virtual URL used for form-based or OpenID-based authentication (by default, `/spring_security_login`) and generates HTML used to render a functional login form.
`o.s.s.web.authentication.www.` `BasicAuthenticationFilter`	It watches for HTTP basic authentication headers and processes them.
`o.s.s.web.savedrequest.` `RequestCacheAwareFilter`	It is used after a successful login to reconstitute the user's original request that was intercepted by an authentication request.

`o.s.s.web.servletapi.`
`SecurityContextHolderAwareRequest`
`Filter`
	It wraps the `HttpServletRequest` with a subclass (`o.s.s.web. servletapi. SecurityContextHolderAwareRequest Wrapper`) that extends `HttpServletRequestWrapper`. This provides additional context to request processors further down the filter chain.

`o.s.s.web.authentication.` `AnonymousAuthenticationFilter`	If the user hasn't already been authenticated by the time this filter is called, an authentication token is associated with the request indicating that the user is anonymous.
`o.s.s.web.session.` `SessionManagementFilter`	It handles session tracking based on authenticated principals, helping to ensure that all sessions associated with a single principal are tracked.
`o.s.s.web.access.` `ExceptionTranslationFilter`	This filter handles default routing and delegation of expected exceptions that occur during processing of a secured request.
`o.s.s.web.access.intercept.` `FilterSecurityInterceptor`	It facilitates determination of authorization and access control decisions, delegating decisions on authorization to an `AccessDecisionManager`.

Spring Security has approximately 25 filters in all that can be conditionally applied based on your particular needs to tweak the behavior of user requests. Of course, you can also add your own servlet filters implementing `javax.servlet.Filter` if you desire.

Keep in mind that the previous table has a list of servlet filters that are automatically populated when you have enabled the `auto-config` attribute in your XML configuration file. Many of the filters from the list can be explicitly included or excluded from your configuration through the addition of other configuration directives, as we will see in later chapters.

You may also configure the filter chain entirely from scratch. Although this is tedious, as there are many dependencies to wire, it can provide a high level of flexibility in configuration and application suitability matching. We cover the Spring Bean declarations required to perform this setup in Chapter 6.

You may wonder how the `DelegatingFilterProxy` is able to locate the filter chain that's configured by Spring Security. Recall that we needed to give the `DelegatingFilterProxy` a filter name in the `web.xml` file:

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filterclass>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>
```

The name of this filter is no coincidence, and in fact is expected for Spring Security to wire itself to the `DelegatingFilterProxy` appropriately. Unless explicitly configured, the `DelegatingFilterProxy` will look for a configured bean in the Spring `WebApplicationContext` of the same name (as specified in the `filter-name` element). More detail on the configurability of the `DelegatingFilterProxy` is available in the Javadoc for the class.

What does auto-config do behind the scenes?

In Spring Security 3, `auto-config` is used solely for automatic provisioning of following three authentication-related functions:

- HTTP basic authentication
- Form login authentication
- Logout

Be aware that it is also possible to declare these three pieces of functionality using configuration elements, with a higher degree of precision than what `auto-config` provides. We will see these used in later chapters to provide more advanced functionality.

> **auto-config isn't what it used to be!**
>
> In releases of Spring Security prior to version 3, the `auto-config` attribute performed a lot more of the setup than it does now. Much of the configuration that was performed by `auto-config` in Spring Security 2 is simply performed by default, using the `security` namespace style of configuration. Please refer to *Chapter 13, Migration to Spring Security 3,* for additional details on migration concerns from Spring Security 2 to 3.

Besides these authentication-related functions, the remainder of the filter chain is set up for you as soon as you use the `<http>` element!

How users are authenticated?

When a user provides credentials in our login form, these credentials must be verified against a **credential store** in order to proceed any further in our secured system. Verification of the credentials presented involves a series of logical components that are used to encapsulate aspects of the authentication process.

We'll work through the example of our sample username and password login form, referring to interfaces and implementations specific to username and password authentication where appropriate. Keep in mind, however, that the overall architecture of authentication is the same, whether you are authenticating a form-based login request, using an external authentication provider such as **Central Authentication Service** (**CAS**), or whether the user credential store is in a database or in an LDAP directory. We'll see how the concepts we explore in form-based login carry through to these more advanced authentication mechanisms in the second half of this book.

The important interfaces related to authentication interact as shown in the following high-level diagram:
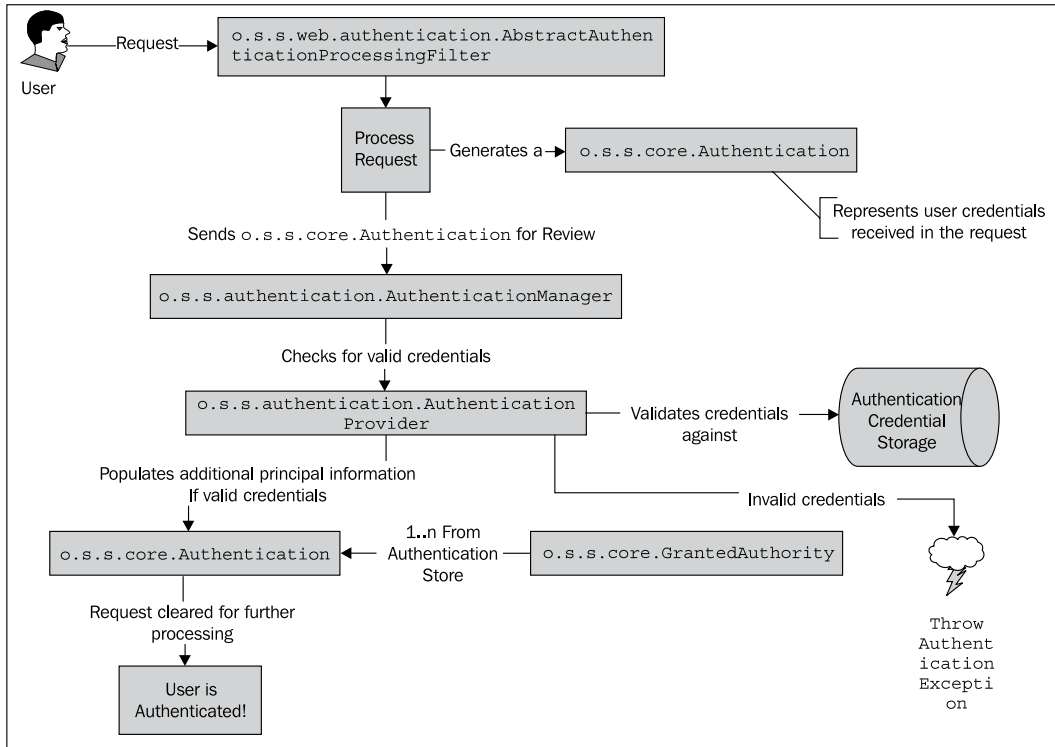
At a high level, you can see that there are three major components that are responsible for much of the heavy lifting:

Interface name	Description / Role
`AbstractAuthenticationProcessingFilter`	It is found in web-based authentication requests. Processes the incoming request containing form POST, SSO information, or other user-provided credentials. Creates a partially filled `Authentication` object to pass user credentials along the chain of responsibility.

Interface name	Description / Role
`AuthenticationManager`	It is responsible for validating the user's credentials, and either throwing specific exceptions (in case of authentication failure), or filling out the `Authentication` object completely, notably including authority information.
`AuthenticationProvider`	It is responsible for providing credential validation to the `AuthenticationManager`. Some `AuthenticationProvider` implementations partially base their acceptance of credentials on a credential store, such as a database.

Implementations of two important interfaces are instantiated by the participants in the authentication chain and used to encapsulate details about an authenticated (or as yet unauthenticated) principal and its authority.

`o.s.s.core.Authentication` is an interface with which you will interact with frequently, because it stores details of the user's identifier (for example, username), credentials (for example, password), and one or more authorities (`o.s.s.core. GrantedAuthority`) which the user has been given. A developer will commonly use the `Authentication` object to retrieve details about the authenticated user, or in a custom authentication implementation he/she will augment the `Authentication` object with additional application-dependent information.

The following methods are available on the `Authentication` interface:

Method signature	Description
`Object getPrincipal()`	It returns the unique identifier of the principal (for example, a username).
`Object getCredentials()`	It returns the credentials of the principal.
`List<GrantedAuthority> getAuthorities()`	It returns the set of authorities that the principal has, as determined by the authentication store.
`Object getDetails()`	It returns provider-dependent details about the principal.

You'll probably note, with concern, that the `Authentication` interface has several methods that simply return `java.lang.Object`. This can make it quite difficult to know at compile-time what type of object you are actually getting back from a method call on the `Authentication` object.

Take note that `AuthenticationProvider` isn't directly used or referenced by the `AuthenticationManager` interface. However, Spring Security ships with only one concrete implementation (plus a subclass) of `AuthenticationManager`, `o.s.s.authentication.ProviderManager`, which uses one or more `AuthenticationProvider` implementations as described. As the use of `AuthenticationProvider` is so prevalent and well-integrated into the `ProviderManager`, it's important to review how it works in the most common scenarios with basic configuration.

Let's get into a little more detail and look specifically at the classes involved in the processing of a web-based username and password authentication request:
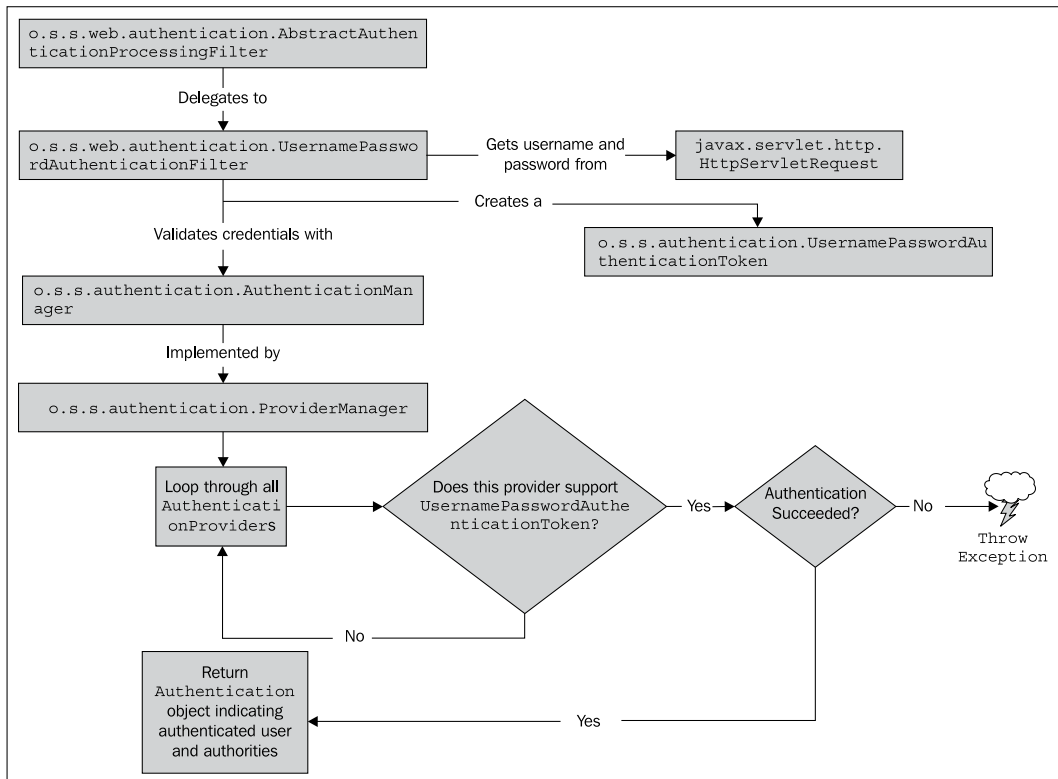
Let's take the abstract workflow seen in the high-level diagram and mentally bring it forward to work through this concrete implementation of form-based authentication. You can see that the `UsernamePasswordAuthenticationFilter` is responsible (through delegation from its abstract superclass) for creating the `UsernamePasswordAuthenticationToken` (an implementation of the `Authentication` interface), and partially populating it based on information in the `HttpServletRequest`. But where does it get the username and password from?

What is spring_security_login and how did we get here?

You may have noticed that when you attempted to hit the home page of our JBCP Pets store, you were redirected to the URL `http://localhost:8080/JBCPPets/spring_security_login`:
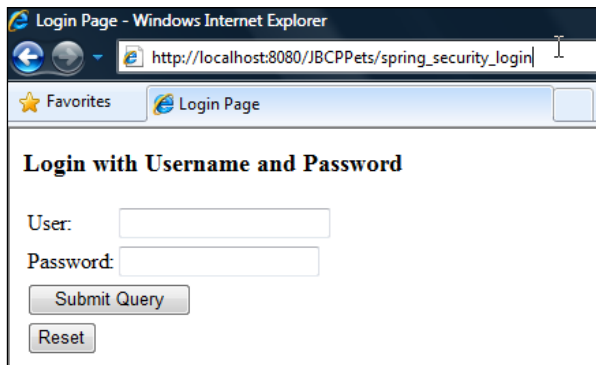
The `spring_security_login` portion of the URL represents a default login page as named in the `DefaultLoginPageGeneratingFilter` class. We can use configuration attributes to adjust the name of this page to be something unique to our application.

> **It's a good idea to change the default value of the login page URL.**
>
> Not only would the resulting URL be more user- or search-engine friendly, it'll disguise the fact that you're using Spring Security as your security implementation. Obscuring Spring Security in this way could make it harder for malicious hackers to find holes in your site in the unlikely event that a security hole is discovered in Spring Security. Although security through obscurity does not reduce your application's vulnerability, it does make it harder for standardized hacking tools to determine what types of vulnerabilities you may be susceptible to.
>
> Watch out, though – this isn't the only place where the name "spring" will show up in the URL! We'll explore this in detail in a later chapter.

Let's look at the HTML source of this form (stripping out table layout information) to figure out what the `UsernamePasswordAuthenticationFilter` is expecting:

```
<form name='f' action='/JBCPPets/j_spring_security_check'
                method='POST'>
  User:<input type='text' name='j_username' value=''>
  Password:<input type='password' name='j_password'/>
  <input name="submit" type="submit"/>
  <input name="reset" type="reset"/>
</form>
```

We can see that there are unusual names (`j_username` and `j_password`) assigned to the two form fields for username and password, and the form action `j_spring_security_check` isn't something that we've configured at all. Where do these come from?

The form field names are specified by `UsernamePasswordAuthenticationFilter` itself, and take their cue from the Java EE Servlet 2.x specification (in section *SRV.12.5.3*), which requires that login forms use these specific field names, plus the special form action `j_security_check`. This design pattern was intended to allow authors of Java EE servlet-based applications to tie into the servlet container's security configuration in a standard way.

As our application is not utilizing the security component of our host servlet container, we could explicitly configure the `UsernamePasswordAuthenticationFilter` to expect form fields with different names. This particular configuration change is more complicated than you might expect; so for now, we'll trace back the lineage of `UsernamePasswordAuthenticationFilter` to see how it arrived in our configuration in the first place (although we do cover this configuration in Chapter 6).

The `UsernamePasswordAuthenticationFilter` is configured through the use of the `<form-login>` sub-element of the `<http>` configuration directive. As we mentioned earlier in this chapter, the `auto-config` attribute that we set will automatically add `<form-login>` capability to your application if you haven't explicitly included the directive. As you may guess, the `j_spring_security_check` URL doesn't map to anything physical in our application. This is a special URL that is watched for by the `UsernamePasswordAuthenticationFilter` to handle form-based login. In fact, there are several of these special URLs that cover specific global behavior in Spring Security. You'll find a table of these URLs in *Appendix*, *Additional Reference Material*.

Where do the user's credentials get validated?

In our simple three-step configuration file, we used an in-memory credential store to get up and running quickly:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider>
    <user-service>
      <user authorities="ROLE_USER" name="guest" password="guest"/>
    </user-service>
  </authentication-provider>
</authentication-manager>
```

We didn't wire this `AuthenticationProvider` to any explicit implementation, and we see once again that the `security` namespace handler performs a lot of rote configuration work on our behalf. Remember that the default implementation of the `AuthenticationManager` supports configuration of one or more `AuthenticationProvider` implementations. The `<authentication-provider>` declaration will by default instantiate an out of the box implementation known as `o.s.s.authentication.dao.DaoAuthenticationProvider`. The declaration of the `<authentication-provider>` element will also automatically wire this `AuthenticationProvider` to the configured (or, in our case, automatically configured) `AuthenticationManager`.

The `DaoAuthenticationProvider` is an `AuthenticationProvider` that provides a thin wrapper to implement the `AuthenticationProvider` interface and then delegates to an implementation of the `o.s.s.core.userdetails.UserDetailsService` interface. The `UserDetailsService` is responsible for returning an implementation of `o.s.s.core.userdetails.UserDetails`.

If you review the Javadoc for `UserDetails`, you'll notice that it looks strikingly similar to the `Authentication` interface we reviewed earlier. Don't get confused, although they have a lot of overlap in method names and capabilities, they have quite different purposes:

Interface	Purpose
Authentication	It stores the identity of the principal, their password, and information about the **context** of the authentication request. It can also contain post-authentication information about the user (which may include an instance of `UserDetails`). Typically not extended, except to support requirements of specialized types of authentication.
UserDetails	Intended to store a profile of the principal, including their name, e-mail, phone number, and so on. Typically extended to support business requirements.

Our declaration of the `<user-service>` subelement triggered a configuration of the `o.s.s.core.userdetails.memory.InMemoryDaoImpl` implementation of the `UserDetailsService`. As you'd anticipate, this implementation stores the users configured in the XML security configuration file in a memory-resident data store. The configuration of this service supports other attributes allowing accounts to be disabled or locked as well.

Let's visually review how the components of the `DaoAuthenticationProvider` interact to provide authentication support to the `AuthenticationManager`:
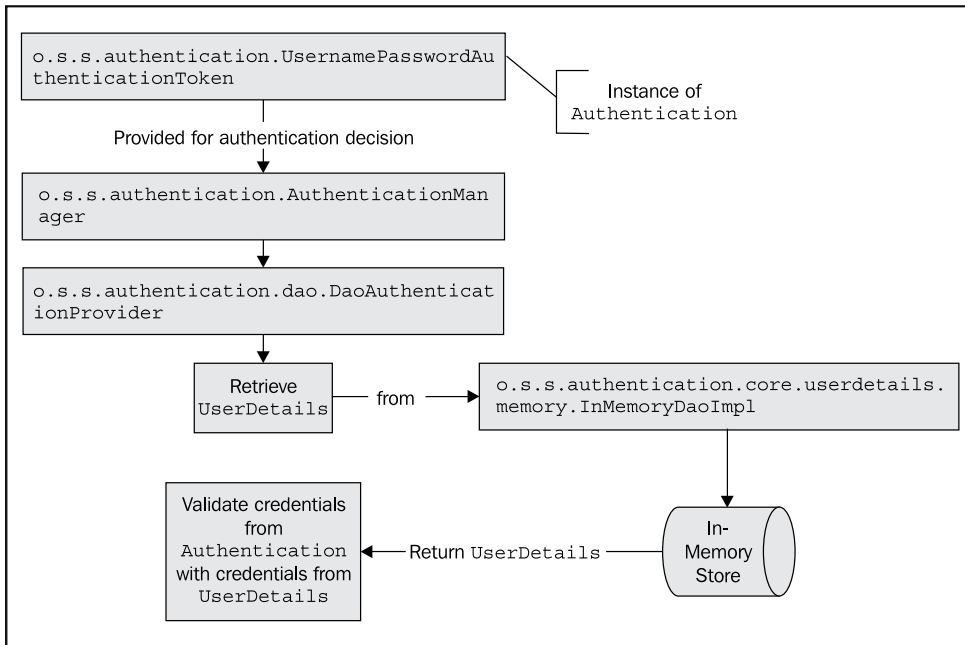
As you may imagine, authentication is extremely configurable. Most Spring Security examples use either the in-memory user credentials store or the JDBC (in-database) credentials store. We've already identified that modifying the JBCP Pets application to store credentials in the database is a good idea, and we'll be handling this configuration change a bit later on, in Chapter 4.

When good authentication goes bad?

Spring Security makes smart use of expected (application) exceptions to indicate conditions whose handling is well-defined. Although you may not directly interact with these exceptions on a day-to-day basis when working with Spring Security, knowing about them and why they are thrown can prove to be very useful in debugging problems, or just understanding the flow of your application.

All authentication-related exceptions extend from the base class `o.s.s.core.AuthenticationException`. In addition to supporting the standard exception functionality, `AuthenticationException` contains two member fields that can help in debugging failures or reporting helpful messages to our users:

- `authentication`: Stores the Authentication instance associated with the authentication request

- `extraInformation`: Stores additional information that may be exception specific. For example, `UsernameNotFoundException` stores the failed username in this field.

We've listed some of the most common exceptions in the following table. The full list of authentication exceptions is provided in *Appendix*, *Additional Reference Material*.

Exception class	When is it thrown?	extraInformation Content
BadCredentialsException	If no username was provided or if the password didn't match the username in the authentication store.	UserDetails
LockedException	If the user's account has been indicated as locked.	UserDetails
UsernameNotFoundException	If the username wasn't found in the authentication store or if the username has no GrantedAuthority assigned.	String (containing the username)

These exceptions and others propagate up the filter chain, and are typically expected and handled gracefully by request-processing filters, either by redirecting a user to an appropriate page (login or access denied), or by returning appropriate HTTP status codes, such as HTTP 403 (Access Denied).

How requests are authorized?

The final servlet filter in the default Spring Security filter chain, `FilterSecurityInterceptor,` is the filter responsible for coming up with a decision on whether or not a particular request will be accepted or denied. At this point the `FilterSecurityInterceptor` filter is invoked, the principal has already been authenticated, so the system knows that they are valid users. Remember that the `Authentication` interface specifies a method (`List<GrantedAuthority> getAuthorities()`), which returns a list of authorities for the principal. The authorization process will use the information from this method to determine, for a particular request, whether or not the request should be allowed.

Remember that authorization is a binary decision—a user either has access to a secured resource or he does not. There is no ambiguity when it comes to authorization.

Smart object-oriented design is pervasive within the Spring Security framework, and authorization decision management is no exception. Recall from our discussion earlier in the chapter that a component known as the **access decision manager** is responsible for making authorization determinations.

In Spring Security, the `o.s.s.access.AccessDecisionManager` interface specifies two simple and logical methods that fit sensibly into the processing decision flow of requests:

- `supports`: This logical operation actually comprises two methods that allow the `AccessDecisionManager` implementation to report whether or not it supports the current request.

- `decide`: This allows the `AccessDecisionManager` to verify, based on the request context and security configuration, whether access should be allowed and the request accepted. `decide` actually has no return value, and instead reports denial of a request by throwing an exception to indicate rejection.

Much like the use of `AuthenticationException` and subclasses that capture expected error conditions during the authentication process, specific types of exceptions can further dictate the action to be taken by the application to resolve authorization decisions. `o.s.s.access.AccessDeniedException` is the most common exception thrown in the area of authorization and merits special handling by the filter chain. We'll review this in detail when we dive into advanced configuration in Chapter 6.

The implementation of `AccessDecisionManager` is completely configurable using standard Spring Bean binding and references. The default `AccessDecisionManager` implementation provides an access granting mechanism based on `AccessDecisionVoter` and vote aggregation.

A **voter** is an actor in the authorization sequence whose job is to evaluate any or all of the following:

- The context of the request for a secured resource (such as URL requesting IP address)

- The credentials (if any) presented by the user

- The secured resource being accessed

- The configuration parameters of the system, and the resource itself

The `AccessDecisionManager` is also responsible for passing the `access` declaration (referred to in the code as implementations of the `ConfigAttribute` interface) of the resource being requested to the voter. In the case of web URLs, the voter will have information about the `access` declaration of the resource. If we look at our very basic configuration file's URL intercept declaration, we'll see `ROLE_USER` being declared as the access configuration for the resource the user is trying to access:

```
<intercept-url pattern="/*" access="ROLE_USER"/>
```

Based on the voter's knowledge, it will make a decision about whether the user should have access to the resource or not. Spring Security allows the voter to make one of three decisions, whose logical definition is mapped to constants in the `o.s.s.access.AccessDecisionVoter` interface:

Decision Type	Description
Grant (`ACCESS_GRANTED`)	The voter recommends giving access to the resource.
Deny (`ACCESS_DENIED`)	The voter recommends denying access to the resource.
Abstain (`ACCESS_ABSTAIN`)	The voter abstains (does not make a decision) on access to the resource. This may happen for a number of reasons, such as: • The voter doesn't have conclusive information • The voter can't decide for a request of this type

As you may have guessed from the design of access decision-related objects and interfaces, this portion of Spring Security has been designed so that it can be applicable to authentication and access control scenarios that aren't exclusively in the web domain. We'll encounter voters and access decision managers when we look at method-level security in *Chapter 5*, *Fine-Grained Access Control*.

When we put this all together, the overall flow of the "default authentication check for web requests" is similar to the following diagram:
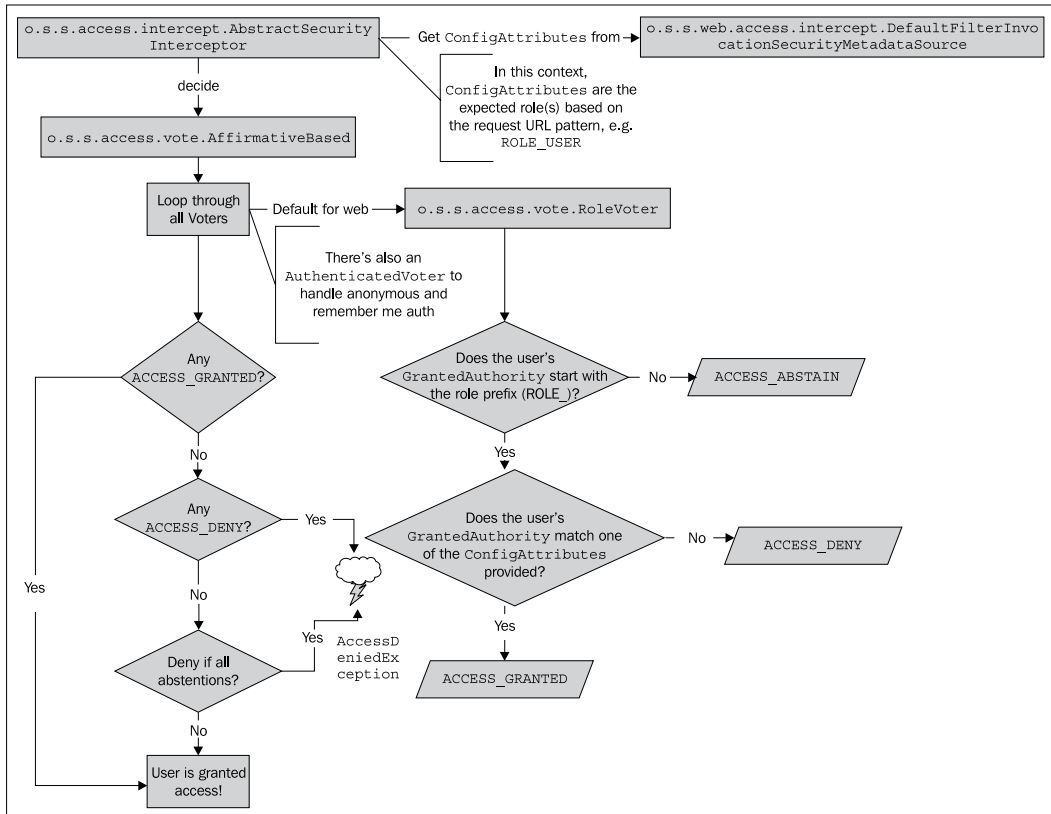
We can see that the abstraction of the `ConfigAttribute` allows for data to be passed from the configuration declarations (retained in the `DefaultFilterInvocationSecurityMetadataSource`) to the voter responsible for acting on the `ConfigAttribute` without any intervening classes needing to understand the contents of the `ConfigAttribute`. This separation of concerns provides a solid foundation for building new types of security declarations (such as the declarations we will see with method security) while utilizing the same access decision pattern.

Configuration of access decision aggregation

Spring Security does actually allow configuration of the `AccessDecisionManager` in the `security` namespace. The `access-decision-manager-ref` attribute on the `<http>` element allows you to specify a Spring Bean reference to an implementation of `AccessDecisionManager`. Spring Security ships with three implementations of this interface, all in the `o.s.s.access.vote` package:

Class name	Description
AffirmativeBased	If any voter grants access, access is immediately granted, regardless of previous denials.
ConsensusBased	The majority vote (grant or deny) governs the decision of the `AccessDecisionManager`. Tiebreaking and handling of empty votes (containing only abstentions) is configurable.
UnanimousBased	All `Voters` must grant access, otherwise access is denied.

Configuring to use a UnanimousBased access decision manager

If we want to modify our application to use the `UnanimousBased` access decision manager, we'd require two modifications. Let's add the `access-decision-manager-ref` attribute to the `<http>` element:

```
<http auto-config="true"
      access-decision-manager-ref="unanimousBased"
>
```

This is a standard Spring Bean reference, so this should correspond to the `id` attribute of a bean. We'll go on and declare the bean (in `dogstore-base.xml`) now, with the same ID we referenced:

```
<bean class="org.springframework.security.access.vote.UnanimousBased"
      id="unanimousBased">
  <property name="decisionVoters">
    <list>
      <ref bean="roleVoter"/>
      <ref bean="authenticatedVoter"/>
    </list>
  </property>
</bean>
<bean class="org.springframework.security.access.vote.RoleVoter"
id="roleVoter"/>
<bean class="org.springframework.security.access.vote.
AuthenticatedVoter" id="authenticatedVoter"/>
```

You may be wondering what the `decisionVoters` property is about. This property is auto-configured until we declare our own `AccessDecisionManager`. The default `AccessDecisionManager` requires us to declare the list of voters who are consulted to arrive at the authentication decisions. The two voters listed here are the defaults supplied by the `security` namespace configuration.

Unfortunately, Spring Security doesn't come supplied with a wide variety of voters, but it is trivial to implement the `AccessDecisionVoter` interface and add our own implementation. We'll see an example of this in Chapter 6.

The two voter implementations that we reference here do the following:

Class Name	Description	Example
`o.s.s.access. vote.RoleVoter`	Checks that the user has the `GrantedAuthority` matching the declared role. Expects the `access` attribute to define a comma-delimited list of `GrantedAuthority` names. The `ROLE_` prefix is expected, but optionally configurable.	`access="ROLE_ USER,ROLE_ ADMIN"`
`o.s.s.access. vote.AuthenticatedVoter`	Supports special declarations allowing wildcard match: • `IS_AUTHENTICATED_ FULLY` – allows access if fresh username and password were supplied • `IS_AUTHENTICATED_ REMEMBERED` – allows access if the user has authenticated with the remember me functionality • `IS_AUTHENTICATED_ ANONYMOUSLY` – allows access if the user is anonymous	`access=" IS_ AUTHENTICATED_ ANONYMOUSLY"`

Access configuration using spring expression language

An alternative method to the standard role-based voting mechanism implemented by `RoleVoter` is the use of **Spring Expression Language** (**SpEL**) expressions to define arbitrarily complex rules for voting. The straightforward way to implement this feature is to add the `use-expressions` attribute to the `<http>` configuration element:

```
<http auto-config="true"
      use-expressions="true">
```

This addition will modify the behavior of the `access` attribute on the URL intercept rule declarations to expect an SpEL expression. SpEL expressions allow for the use of expression language specifications of access criteria. Instead of simple strings such as `ROLE_USER`, the configuration file can specify expressions that invoke method calls, reference system properties, compute values, and much more.

The SpEL syntax will look familiar to users of other expression-type languages, such as **Object Graph Notation Language** (**OGNL**) used in the Tapestry framework, among others), and JSP or JSF Unified Expression Language (used in JSP and/or JSF development). The syntax is comprehensive enough, so covering the whole language is quite beyond the scope of this book; however, we'll work through several examples that should provide you with some concrete help in constructing expressions.

An important point to note is that if you enable the SpEL expression-based access specifications by setting the `use-expressions` attribute, you will disable the automatic configuration of the `RoleVoter`, which understands declarations of roles, like we saw in our simple configuration:

```
<intercept-url pattern="/*" access="ROLE_USER"/>
```

This means that your access declarations must change if you want to filter access solely by role. Fortunately, this was anticipated, and an SpEL-bound method `hasRole` is available to check roles. If we rewrote our sample configuration file to use expressions, it would look like this:

```
<http auto-config="true" use-expressions="true">
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
</http>
```

As you might expect, the SpEL handling is supplied by a different `Voter` implementation, `o.s.s.web.access.expression.WebExpressionVoter`, which understands how to evaluate the SpEL expressions. The `WebExpressionVoter` relies on an implementation of the `o.s.s.web.access.expression.WebSecurityExpressionHandler` interface for this purpose. The `WebSecurityExpressionHandler` is responsible both for evaluating the expressions, as well as supplying the security-specific methods that are referenced in the expressions. The default implementation of this interface exposes methods defined in the `o.s.s.web.access.expression.WebSecurityExpressionRoot` class.

The flow and relationship between these classes is shown in the following diagram:

```
┌──────────────────────────────────────────────────────────────────────────┐
│   o.s.s.access.vote.AccessDecisionManager                                  │
│              └────── Gets Vote From ──────┐                                │
│                 o.s.s.web.access.expression.WebExpressi                    │
│                 onVoter                                                     │
│                                                                            │
│                  ├──── Gets SpEL Context From ────┐                        │
│                                          o.s.s.web.access.expression.WebSecurity │
│                                          ExpressionHandler                  │
│                                              Gets SpEL Methods From          │
│                                          o.s.s.web.access.expression.WebSecurity │
│                                          ExpressionRoot                      │
│                           ──── When Done ────                               │
│                                                                            │
│              Evaluates Expression as ──── false ──→ ACCESS_DENY            │
│                    Boolean                                                  │
│                      │                                                      │
│                     true                                                    │
│                 ACCESS_GRANTED                                              │
│  ── Vote Returned to AccessDecisionManager ──                              │
└──────────────────────────────────────────────────────────────────────────┘
```

Methods and pseudo-properties for SpEL access expressions are declared by the public methods provided by the `WebSecurityExpessionRoot` class, and its superclasses.
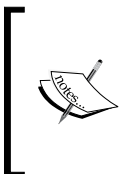
A **pseudo-property** is a method that takes no parameters and complies with JavaBeans naming convention for getters. This allows the SpEL expression to omit the parentheses and is or get prefix on the method. For example, the method `isAnonymous()` can be accessed as the pseudo-property `anonymous`.

The available SpEL methods and pseudo-properties which ship with
Spring Security 3 are shown in the following tables. Note that methods and
properties not marked as "web only" are available for use when securing other types
of resources that utilize SpEL, such as method calls. The examples provided illustrate
the use of the method or property in an `<intercept-url>` `access` declaration.

Method	Web only?	Description	Example

`hasIpAddress (ipAddress)`	Yes	Used to match the IP address of the request to either a specific IP address, or an IP address plus netmask.	`access="hasIpAddress(' 162.79.8.30')"`
			
			 `access="hasIpAddress(' 162.0.0.0/224')"`

`hasRole(role)`	No	Used to match a role with a `GrantedAuthority` (similar to `RoleVoter` declaration).	`access="hasRole('ROLE_ USER')"`
`hasAnyRole(role)`	No	Used to match from a list of roles with the user's `GrantedAuthority`. Users matching any of the roles will be granted access.	`access="hasRole('ROLE_ USER','ROLE_ADMIN')"`

In addition to the methods in the previous table, a series of methods are provided
that can act as properties in the SpEL expressions. These do not require parentheses
or method arguments.

Property	Web only?	Description	Example
`permitAll`	No	Always grants access to any user.	`access="permitAll"`
`denyAll`	No	Always denies access to any user.	`access="denyAll"`
`anonymous`	No	Indicates if the user is anonymous.	`access="anonymous"`

Property	Web only?	Description	Example
`authenticated`	No	It indicates if the user has been authenticated.	`access="authenticated"`
`rememberMe`	No	It indicates if the user has been authenticated with the remember me feature.	`access="rememberMe"`
`fullyAuthenticated`	No	It indicates if the user has been authenticated with a full set of credentials during this request.	`access="fullyAuthenticated"`

Remember that voter implementations must return a voting decision (grant, deny, or abstain) based on the context of the request. You may note that `hasRole` sounds like it returns a Boolean response, and in fact this is true. SpEL-based access declarations must consist only of expressions which return a Boolean result. A `true` result means that the voter grants access, and a `false` result means that the voter denies access.

> If you try to return an expression that doesn't evaluate to a Boolean, you'll get an unfriendly exception with a message like this:
>
> ```
> org.springframework.expression.spel.SpelException:
> EL1001E:Type conversion problem, cannot convert from
> class java.lang.Integer to java.lang.Boolean
> ```

In addition, it is not possible for an expression to return a result indicating abstention, unless the access declaration is not a valid SpEL expression, in which case the voter will abstain from voting.

If you don't mind these minor limitations, SpEL based access declarations can provide a flexible method of configuring access decisions.

Summary

This chapter provided us with a solid introduction to the twin security concepts of authentication and authorization. We've:

- Explored the high-level architecture of our secured system
- Used the automatic configuration functionality of Spring Security to secure the JBCP Pets website in three steps
- Reviewed the use and importance of servlet filters in Spring Security
- Examined the key actors in the authentication and authorization processes, including detailed introductions to important objects such as `Authentication` and `UserDetails`
- Performed some configuration to experiment with the SpEL expression language as it relates to specification of access rules

In the next chapter, we'll take the basic username and password authentication to the next level by adding some key features to better integrate the user experience with the flow of the website.

3
Enhancing the User Experience

In this chapter, we'll add functionality to the JBCP Pets store to provide our users with a more pleasant and usable experience while enhancing the website to support many important behaviors of a secure website.

During the course of this chapter, we'll:

- Customize the login and logout pages to your liking, and tie them into standard Spring web MVC controllers
- Enable remember me functionality for user convenience and understand its security implications
- Build user account management features, including change password and forgot password functionality

Customizing the login page

You'll recall that in the previous chapter, we used the baseline `security` namespace configuration functionality of Spring Security. While this provided us with very basic login, authentication, and authorization, it's definitely not production ready. One of the most obvious enhancements we need to make before we show off our progress to our boss is to make the default login page look and act like the rest of our online store.

Recall that the login page currently is similar to the following screenshot:

The automatic configuration didn't provide us with some other important features including the styled login page. We decide to add the following features to the site:

- A login page that has the header, footer, and look-and-feel of the rest of the JBCP Pets site
- A link that allows the user to log out
- A page that allows the user to change his or her password

The flow of the login and logout functionality should be similar to the following diagram when we're done:
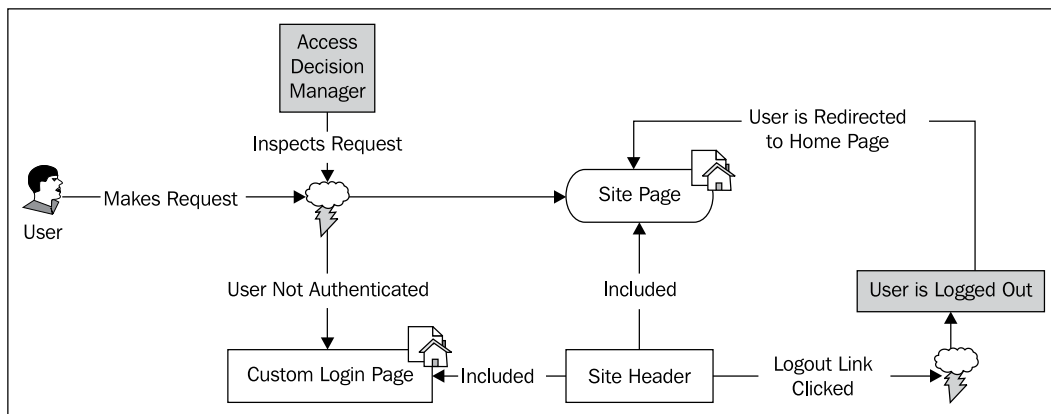
We'll work our way through developing this site structure through a series of exercises. As we work through the development of true login and logout functionality, we'll explain and explore what we're doing, so that when the time comes to extend the basic functionality of the site, we'll have a clear understanding of what we're building!

Implementing a custom login page

At first, we'll need to replace the default Spring Security login page with a login page that integrates into our online store. The desired login flow should be as follows:
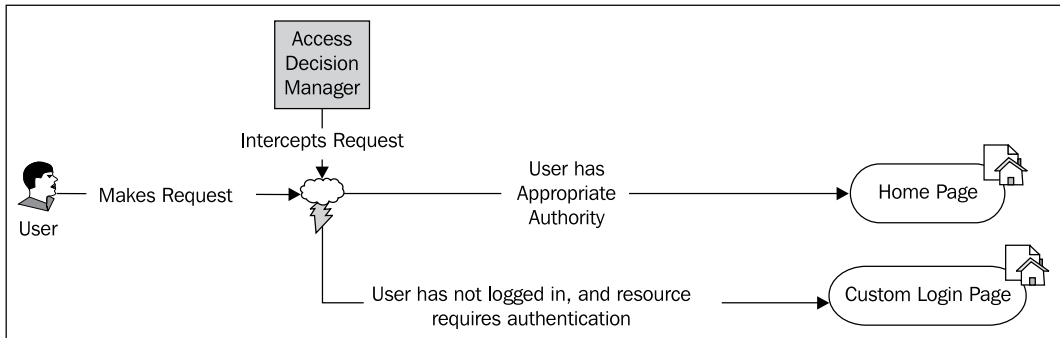
Implementing the login controller

We'll need to add a new Spring MVC controller to handle first the login and then the logout functionality. The JBCP Pets site uses the Spring MVC annotation-based mechanism for configuration of controllers and website paths and resources. Let's create a controller called `LoginLogoutController` in `com.packtpub.springsecurity.web.controller` with the following content:

```
// imports omitted
@Controller
public class LoginLogoutController extends BaseController{
    @RequestMapping(method=RequestMethod.GET,value="/login.do")
    public void home() {
    }
}
```

As you can see, we've added a very simple controller, and mapped its single method to the URL, that is, `/login.do`. This is all we need for now to build the baseline login page under our control, instead of the default Spring Security baseline login page that the baseline configuration gives us. The `BaseController` super class already exists from our work in *Chapter 2, Getting Started with Spring Security*, and provides a convenient place to put methods that can be used from every controller in the application.

Adding the login JSP

The `/login.do` reference will cause the Spring MVC view resolver that we've configured in `WEB-INF/dogstore-servlet.xml` to look for a JSP called `login.jsp` in `/WEB-INF/views`. Let's add a simple JSP with a login form that will be recognized by Spring Security. Remember that we learned in Chapter 2 that there are 2 important elements of the login form that must be correct in order for the appropriate actions to occur:

- The **form action** must match the action configured in the `UsernamePasswordAuthenticationFilter` servlet filter. By default, this form action is `j_spring_security_check`.

- The form fields for username and password must match the servlet specifications. By default `j_username` and `j_password` are the form field names.

We'll also tie the JSP in to the sitewide header and footer includes (we've included these with the sample code for this chapter, but omitted them from the text, as they aren't pertinent for now). All this leaves us with a fairly simple JSP:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<jsp:include page="common/header.jsp">
  <jsp:param name="pageTitle" value="Login"/>
</jsp:include>
<h1>Please Log In to Your Account</h1>
<p>
  Please use the form below to log in to your account.
</p>
<form action="j_spring_security_check" method="post">
  <label for="j_username">Login</label>:
  <input id="j_username" name="j_username" size="20" maxlength="50"
type="text"/>
  <br />
  <label for="j_password">Password</label>:
  <input id="j_password" name="j_password" size="20" maxlength="50"
type="password"/>
  <br />
  <input type="submit" value="Login"/>
</form>
<jsp:include page="common/footer.jsp"/>
```

Be aware that you must use a form POST, otherwise the login request will be rejected by the `UsernamePasswordAuthenticationFilter`.

Finally, we'll need to modify the automatic configuration settings of Spring Security to refer to our new login page. If you're feeling adventurous at this point, we actually have added a new, working page to our application. Try following along up to this point and hitting the URL `http://localhost:8080/JBCPPets/login.do` to see what happens.

What happened? Did you notice that your request was first intercepted by Spring Security (redirected to `spring_security_login`), and then you saw the form? This is because Spring Security still refers to the default login page generated by the `DefaultLoginPageGeneratingFilter` class. Once you get past the default login page generated by this filter, you'll be presented with your new, customized login page. The final step is to remove the default page, and use our login form as the login page.

Configuring Spring Security to use our Spring MVC login page

On first glance, it seems like all we'd need to do is configure the `<form-login>` element in the Spring Security configuration file to add the `login-page` directive, similar to the following:

```
<http auto-config="true" use-expressions="true">
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
  <form-login login-page="/login.do" />
</http>
```

Now try starting up the application and hitting the home page (`http://localhost:8080/JBCPPets/home.do`). If you're using Internet Explorer, you'll notice that the page never renders at all, but the browser simply spins and spins. Let's switch to Mozilla Firefox and try the same URL. With Firefox, you'll get more information as to what's going on:
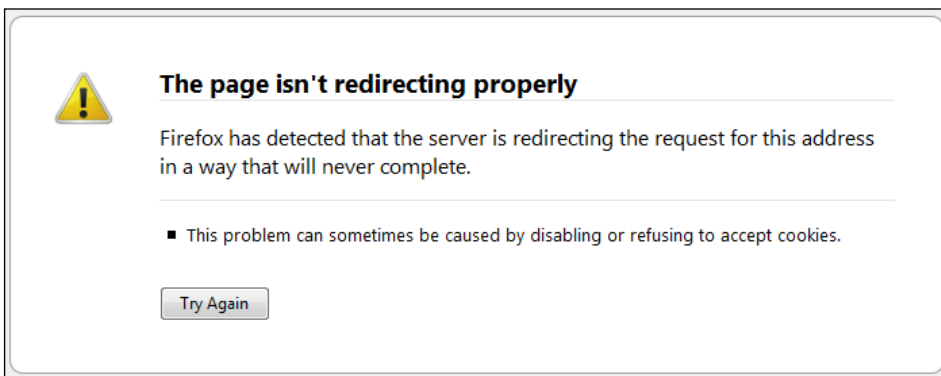
The problem here is our rule for URL interception:

```
<intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
```

This will require the user to have ROLE_USER to access any page matching the URL /*—this matches every page in the application, including our new login page!

The following diagram illustrates what's happening:
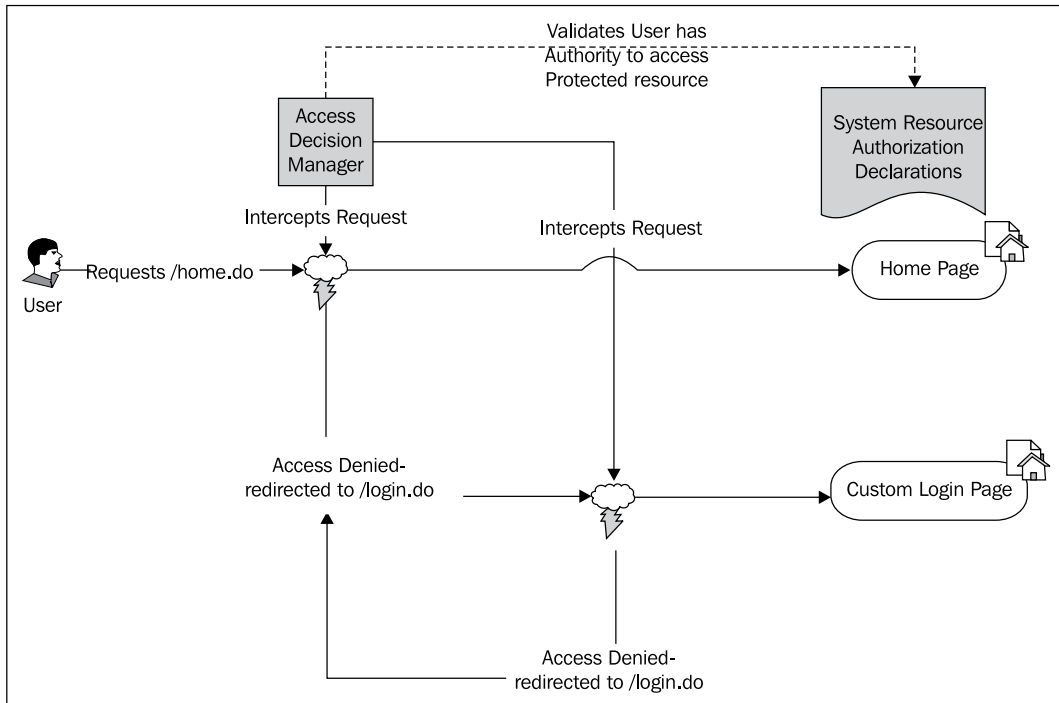
We'll need to update our authorization rule set to allow anonymous users to access our login page.

For any given URL request, Spring Security evaluates authorization rules in top to bottom order. The first rule matching the URL pattern will be applied. Typically, this means that your authorization rules will be ordered starting from most-specific to least-specific order. It's important to remember this when developing complicated rule sets, as developers can often get confused over which authorization rule takes effect. Just remember the top to bottom order, and you can easily find the correct rule in any scenario!

As we're adding a more specific rule, we'll put it at the top of the list. We'll end up with the following rule set:

```
<intercept-url pattern="/login.do" access="permitAll"/>
<intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
```

This will result in the behavior we want, enabling any user to access the login page, while still restricting the rest of the site to authenticated users only. We're done exploring login for now. Let's look at what we need to do to add logout functionality.

Understanding logout functionality

Logout is the term for a user-initiated action that results in the user's secure session being invalidated. Typically, the user will also be redirected to a non-secured area of the site after they have chosen to log out. Let's add a **Log Out** link to the site header, and again browse to the site to explore how and why it works after we're done.

Adding a Log Out link to the site header

As we discussed in Chapter 2, Spring Security is aware of several special URLs that serve as watch points for behavior to be triggered in one or more servlet filters in the filter chain. The signature URL for logging the user out is /j_spring_security_logout. Adding a logout link is really as simple as putting an anchor tag with the appropriate href in the header.jsp file:

```
<c:url value="/j_spring_security_logout" var="logoutUrl"/>
<li><a href="${logoutUrl}">Log Out</a></li>
```

If you reload the site and click the **Log Out** link, you will find that you'll be back at the login form. Have fun logging in and logging out for a while!

> **Using the JSTL URL tag to handle relative URLs**
>
> We use the JSTL core library's url tag to ensure that URLs we provide resolve correctly in the context of our deployed web application. The url tag will resolve URLs provided as relative URLs (starting with a /) to the root of the web application. You may have seen other techniques to do this using JSP expression code (<%= request.getContextPath() %>), but the JSTL url tag allows you to avoid inline code!

Let's see what's happening behind the scenes of this very simple operation.

How logout works

What really happened when we clicked on the Log Out link?

Remember that every request for a URL goes through the entire Spring Security filter chain, before being resolved to a servlet request. So, although the URL request for `/j_spring_security_logout` doesn't correspond to a JSP in our system, it doesn't have to be a real JSP or Spring MVC destination in order to be handled. These types of URLs are often referred to as virtual URLs.

The URL request for `/j_spring_security_logout` is intercepted by the `o.s.s.web.authentication.logout.LogoutFilter`. One of the many filters in the default Spring Security filter chain, the `LogoutFilter` looks for this particular virtual URL and takes action.

Let's quickly examine the configuration that the `security` namespace provides us with regards to default logout functionality:

```
<http auto-config="true" use-expressions="true">
  <logout invalidate-session="true"
      logout-success-url="/"
      logout-url="/j_spring_security_logout"/>
</http>
```

This baseline configuration will look for the logout URL specified in the `logout-url` attribute and log the user out. Logging the user out involves the following 3 steps:

1. Invalidating the HTTP session (if `invalidate-session` is set to `true`).
2. Clearing the `SecurityContext` (effectively logging the user out).
3. Redirecting to the URL specified in `logout-success-url`.

The following diagram illustrates how the logout process works:
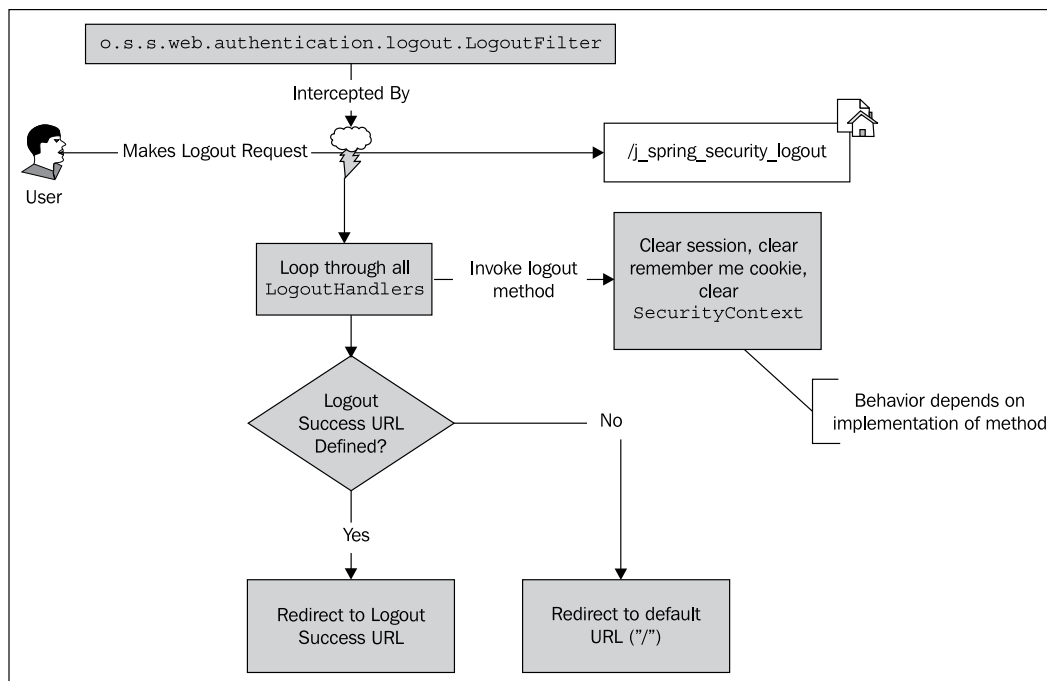
`o.s.s.web.authentication.logout.LogoutHandler` is an interface whose implementation classes can be invoked upon user logout by the `LogoutFilter`. It is possible (although complex) to implement your own `LogoutHandler` that will be tied into the `LogoutFilter` lifecycle. The default set of `LogoutHandlers` that are configured with the `LogoutFilter` are responsible for clearing the session and cleaning up the remember me feature so that the user's session is now operating with no remaining authentication associated. Finally, the redirection to a URL after logout is performed by a default implementation of the interface `o.s.s.web. authentication.logout.LogoutSuccessHandler`. This default implementation simply redirects to the success URL configured (the default is /), but can be updated to perform anything else that your application needs to be done after the user is logged out. It is important to note that logout handlers should not throw exceptions, as it's important for all of them to execute to avoid potential inconsistency in the user's secured session. Take care that exceptions are properly handled and logged when implementing your own logout handlers.

Changing the logout URL

Let's test overriding the default logout URL to provide a simple example of modifying the automatically configured behavior. We'll change the logout URL to /logout.

Modify the dogstore-security.xml file to contain the <logout> element similar to the following code:

```
<http auto-config="true" use-expressions="true">
...
..<logout invalidate-session="true"
      logout-success-url="/"
      logout-url="/logout"/>
</http>
```

Modify the /common/header.jsp file to change the logout link's href attribute to match the new URL:

```
<c:url value="/logout" var="logoutUrl"/>
<li><a href="${logoutUrl}">Log Out</a></li>
```

Restart the application and try it out! You'll observe that instead of /j_spring_security_logout, the /logout URL will be used to log the user out. You may also notice that if you try /j_spring_security_logout, you'll get a Page not Found (404) error, because the URL doesn't correspond to an actual servlet resource and is no longer handled by a request filter.

Logout configuration directives

The <logout> element contains additional configuration directives that allow for more sophisticated logout functionalities that are explained in the following table:

Attribute	Description
invalidate-session	If true, the user's HTTP session will be invalidated when they log out. In some cases, this isn't desired (for example, when a user has a shopping cart).
logout-success-url	The URL to which the user will be redirected when they log out. Defaults to /. This is handled as a HttpServletResponse.redirect.
logout-url	The URL that the LogoutFilter reads (we changed this in the exercise).
success-handler-ref	A bean reference to an implementation of LogoutSuccessHandler.

Remember me

A convenient feature to offer frequent users of the website is a remember me feature. This feature allows a returning user to be remembered through a simple cookie, encrypted and stored on the user's browser. If Spring Security recognizes that the user is presenting a remember me cookie, the user will be automatically logged in to the application, and will not need to enter a username or password.

Unlike the other features we've discussed up to this point, the remember me feature is not automatically configured when we utilize the `security` namespace style of configuration. Let's try out this feature and see how it affects the flow of the login experience.

Implementing the remember me option

Completing this exercise will allow us to provide a simple user memory functionality to the pet store.

Modify the `dogstore-security.xml` configuration file to add the `<remember-me>` declaration. Set the key attribute to `jbcpPetStore`:

```
<http auto-config="true" use-expressions="true" access-decision-
manager-ref="affirmativeBased">
…
    <remember-me key="jbcpPetStore"/>
    <logout invalidate-session="true" logout-success-url="/" logout-
url="/logout"/>
  </http>
```

If we try the application now, we'll see nothing different in the flow. This is because we also need to add a field to the login form allowing the user to opt in to this functionality. Edit the `login.jsp` file to add a checkbox similar to the following:

```
<input id="j_username" name="j_username" size="20" maxlength="50"
type="text"/>
<br />
<input id="_spring_security_remember_me" name="_spring_security_
remember_me" type="checkbox" value="true"/>
<label for="_spring_security_remember_me">Remember Me?</label>
<br />
<label for="j_password">Password</label>:
```

When we next log in, if the **Remember Me** box is checked, a remember me cookie is set in the user's browser.

If the user then closes their browser and reopens it to an authenticated page on the JBCP Pets website, they won't be presented with the login page again. Try it yourself now—log in with the **Remember Me** option checked, bookmark the home page, then restart the browser and access the home page again. You'll see that you're immediately logged in successfully without needing to supply credentials again.

Sophisticated users wishing to exercise this functionality can also use browser plug-ins to manipulate (remove) session cookies, such as Firecookie (`http://www.softwareishard.com/blog/firecookie/`). This can often save time and annoyance during development and verification of this type of feature on your site.

How remember me works

The remember me feature sets a cookie on the user's browser containing a Base64 encoded string with the following pieces:

- The user's username
- An expiration date/time
- An MD5 hash of the expiration date/time, username, and password
- The application key defined in the key attribute of the `<remember-me>` element

These bits are combined into a single cookie value that is stored on the browser for later use.

MD5 is one of several well-known **cryptographic hash algorithms**. Cryptographic hash algorithms compute a compact and unique text representation of input data with arbitrary length, called a **digest**. This digest can be used at other times to verify that unknown input precisely matches the input used to generate the hash, without requiring the availability of the original input. The following diagram illustrates how this works:
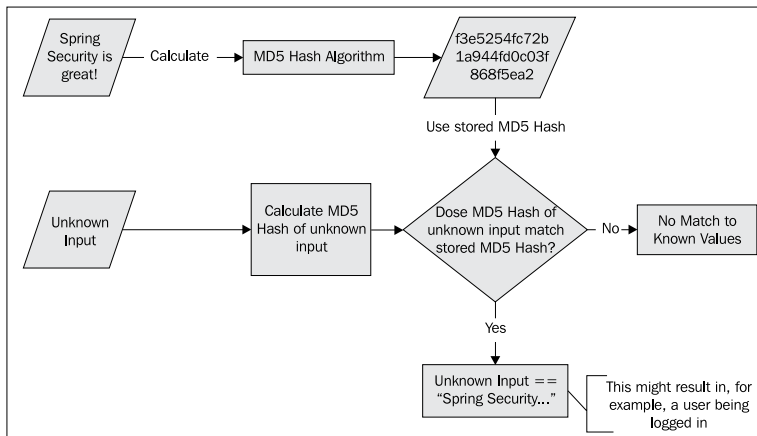
As you can see, the unknown input can be verified against the stored MD5 hash, and the conclusion can be drawn that the unknown input matches the known input. A key difference between digest and encryption algorithms is that it is very difficult to reverse engineer the original data from the digest value. This is because the digest represents only a summary, or fingerprint, of the original data, and not the entire data itself (which may be quite large).

Although it is impossible to decode the encrypted data, MD5 is vulnerable to several types of attacks, including the exploit of weaknesses in the algorithm itself and **rainbow table attacks**. Rainbow tables typically contain the pre-computed hashes of millions of input values. This allows attackers to look for the hash value in the rainbow table and determine the actual (unhashed) value. We'll see a method of combating this in *Chapter 4, Securing Credential Storage*, when we review password security.

With regards to remember me cookies, we'll see when we review the composition of the remember me cookie that the cookie ingredients are complex enough that it would be somewhat hard for an attacker to create a hacked cookie. In Chapter 4, we'll learn another technique to further secure remember me itself from malicious attackers.

The cookie expires based on an expiration period of configurable length. If the user revisits our site before the cookie expires, the cookie is presented to the application as with any other cookie which the application has set.

In the case of the remember me cookie, the `o.s.s.web.authentication.` `rememberme.RememberMeAuthenticationFilter` inserted into the filter chain by the `<remember-me>` configuration directive will review the contents of the cookie and use it to authenticate the user if it seems to be an authentic remember me cookie (see the *Is remember me secure?* section later in this chapter for reasons why this is done).

The following diagram illustrates the different components involved in the process of validating a remember me cookie:
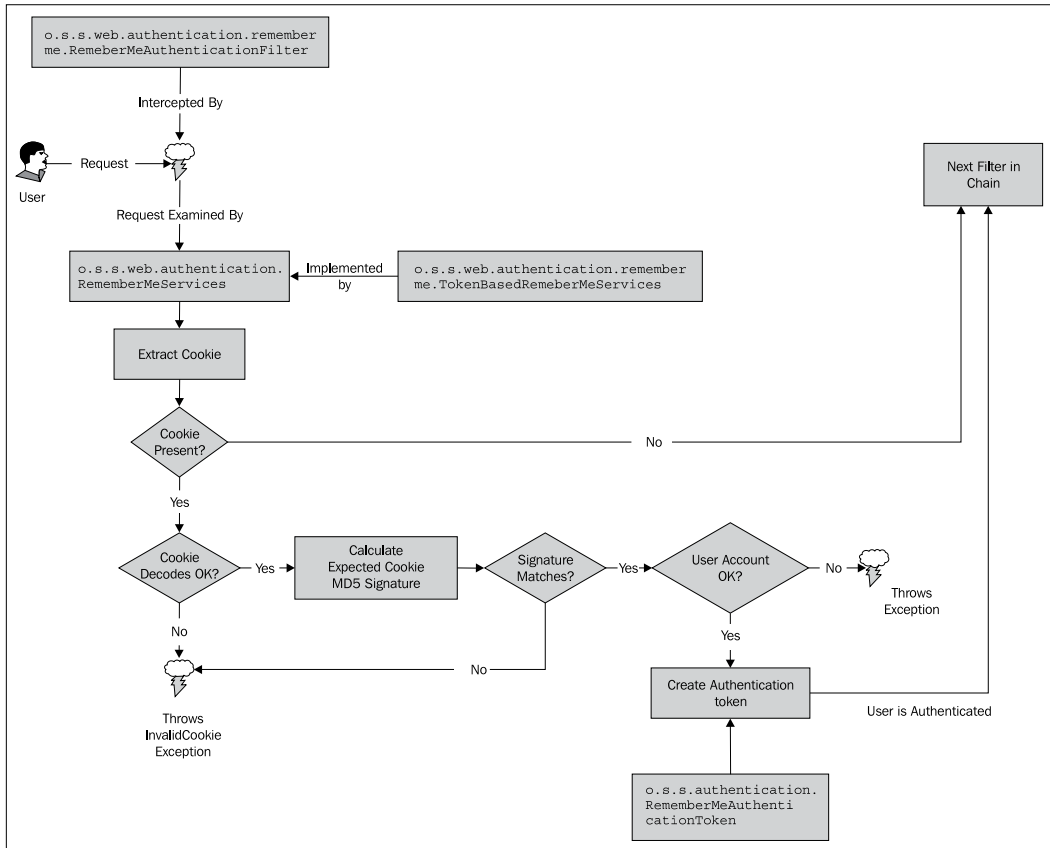
The `RememberMeAuthenticationFilter` is inserted into the filter chain just after the `SecurityContextHolderAwareRequestFilter`, and just before the `AnonymousProcessingFilter`. Just as the other filters in the chain do, the `RememberMeAuthenticationFilter` will also inspect the request, and if it is of interest, action is taken.

As the diagram indicates, the filter is responsible for investigating if the user's browser has supplied a remember me cookie as part of their request. If a remember me cookie is found, it is Base64 decoded, and the expected MD5 hash is calculated based on the username and password presented in the cookie. Once the cookie passes this level of validation, the user has been logged in.

> You have anticipated that if the user changes their username or password, any remember me tokens set will no longer be valid. Make sure that you provide appropriate messaging to users if you allow them to change these bits of their account. In Chapter 4, we will look at an alternative remember me implementation that is reliant only on the username and not on the password.

We see that the `RememberMeAuthenticationFilter` relies on an implementation of `o.s.s.web.authentication.RememberMeServices` to validate the cookie. The same implementation class is also used upon successful form-based login, if it sees a form parameter come along with a login request that has the name `_spring_security_remember_me`. The cookie is securely encoded with information as described earlier, being stored on the browser in Base64 encoding, and containing an MD5 hash including a timestamp and the user's password.

Note that it is still possible to differentiate between users who have been authenticated with a remember me cookie and users who have presented username and password (or equivalent) credentials. We'll experiment with this shortly when we investigate the security of the remember me feature.

Remember me and the user lifecycle

The implementation of `RememberMeServices` is invoked at several points in the user lifecycle (the lifecycle of an authenticated user's session). To assist in your understanding of the remember me functionality, it can be helpful to be aware of the points in time when remember me services are informed of lifecycle functions:

Action	What should happen?
Successful login	Implementation sets a remember me cookie (if the form parameter has been sent).
Failed login	Implementation should cancel the cookie, if it's present.
User logout	Implementation should cancel the cookie, if it's present.

Knowing where and how `RememberMeServices` ties in to the user lifecycle will be important when we begin to create custom authentication handlers, because we need to ensure that any authentication processor treats the `RememberMeServices` consistently, to preserve the usefulness and security of this functionality.

Remember me configuration directives

Two configuration changes are commonly made to alter the default behavior of the remember me functionality:

Attribute	Description
Key	Defines a unique key for remember me cookies associated with our application.
token-validity-seconds	Defines the length of time (in seconds). The remember me cookie will be considered valid for authentication. Also used to set the cookie expiration timestamp.

As you may infer from the discussion of how the cookie contents are hashed, the key attribute is critical to security of the remember me feature. Make sure that the key you choose is likely to be unique to your application, and long enough so that it can't be easily guessed.

Keeping in mind the purpose of this book, we've kept the key values relatively simple, but if you're using remember me in your own application, it's suggested that your key contains the unique name of your application and is at least 36 random characters long. Password generator tools (Search Google for "online password generator") are a great way to get a pseudo-random mix of alphanumeric and special characters to compose your remember me key.

Keep in mind that for applications that exist in multiple environments (such as development, test, production), the remember me cookie value should include this fact as well. This will prevent remember me cookies from inadvertently being used in the wrong environment during testing!

An example key value in a production application might is as shown:

```
jbcpPets-rmkey-paLLwApsifs24THosE62scabWow78PEaCh99Jus
```

The token-validity-seconds attribute is used to set the number of seconds after which the remember me token will not be accepted for the automatic login function, even if it is otherwise a valid token. The same attribute is also used to set the maximum lifetime of the login cookie on the user's browser.

> **Configuration of remember me session cookies**
>
> If `token-validity-seconds` is set to `-1`, the login cookie will be set to a session cookie, which does not persist after the user closes their browser. The token will be valid (assuming the user doesn't close their browser) for a non-configurable length of 2 weeks. Don't confuse this with the cookie that stores your user's session ID—they're two different things with similar names!

There are several other configuration directives related to advanced customization of the remember me functionality. We'll cover some of these in the next set of exercises, and others in *Chapter 6, Advanced Configuration and Extension*, when we deal with advanced authorization techniques.

Is remember me secure?

Any feature related to security that has been added for user convenience has the potential to expose a security risk to our carefully protected site. The remember me feature, in its default form, runs the risk of the user's cookie being intercepted and reused by a malicious user. The following diagram illustrates how this might happen:
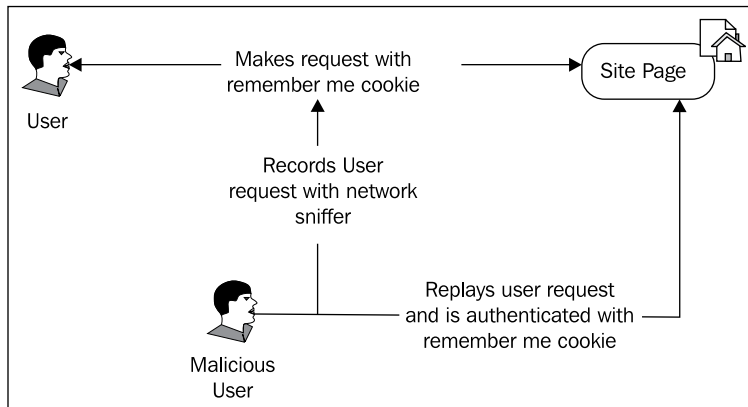
Use of SSL (covered in Chapter 4) and other network security techniques can mitigate this type of attack, but be aware that there are other techniques such as **cross-site scripting** (**XSS**) that could steal or compromise a remembered user session. While convenient for the user, we don't want to risk financial or other personal information being inadvertently changed or possibly stolen if the remembered session is misused.

> Although we don't cover malicious user behavior in detail in this book, when implementing any secured system it is important to understand the techniques employed by users who may be trying to hack your customers or employees. XSS is one such technique, but many others exist. It's highly recommended that you review the **OWASP Top Ten** (`http://www.owasp.org/index.php/Category:OWASP_ Top_Ten_Project`) for a good starting list, and also pick up a web application security reference book, where many of the techniques demonstrated are illustrated to apply to any technology.

One common approach for maintaining the balance between convenience and security is identifying the functional locations in the site where personal or sensitive information could be present. Ensure these locations are protected using authorization that checks not just the user's role, but that they have been authenticated with a full username and password. This can be done using the `fullyAuthenticated` pseudo-property supported by the SpEL expressions for authorization rules, which we covered in Chapter 2.

Authorization rules differentiating remembered and fully authenticated sessions

We'll fully explore advanced authorization techniques later in *Chapter 5, Fine-Grained Access Control*, however, it's important to realize that it's possible to differentiate access rules based on whether an authenticated session was remembered.

Let's assume that the user may look at and modify their "wish list" on the site when they come in with a remembered session. This is similar to behavior found in other major consumer-focused commerce sites, and doesn't present a risk to personally identifiable or financial information (keep in mind that every site is different, and don't blindly apply such rules to your secure site!). Instead, we'll concentrate on protecting user account and ordering functionality. We want to ensure that even remembered users who try to access account information, or place an order, are required to authenticate themselves. Here's how we'd set up the authorization rules:

```
<intercept-url pattern="/login.do" access="permitAll"/>
<intercept-url pattern="/account/*.do"
   access="hasRole('ROLE_USER') and fullyAuthenticated"/>
<intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
```

The existing rules for the login page and ROLE_USER remain unchanged. We've added a rule that requires requests for account information to have the appropriate GrantedAuthority of ROLE_USER, and that the user is fully authenticated, that is, during this authenticated session, they have actually presented a username and password or other suitable credentials. Note the syntax of SpEL logical operators here—and, or, and not are used for logical operators in SpEL. This was thoughtful of the SpEL designers, as the && operator would be awkward to represent in XML!

Keep in mind that if you try this, you'll receive a **403 Access Forbidden** message if you try to access the **My Account** link after using remember me to log in, which we've now suitably protected. This is because the application is still configured with the default AccessDeniedHandler, that is responsible for catching and responding to AccessDeniedException reports. We'll customize this behavior in Chapter 6, when we learn how AccessDeniedException is handled.

> **Checking Full Authentication without Expressions**
>
> If your application does not use SpEL expressions for access declarations, you can still check if the user is fully authenticated by using the IS_AUTHENTICATED_FULLY access rule (For example,. access=" IS_AUTHENTICATED_FULLY"). Be aware, however, that standard role access declarations aren't as expressive as SpEL ones, so you will have trouble handling complex boolean expressions.

Error handling not withstanding, you can see that this approach combines the usability enhancements of the remember me feature with additional level of security requiring a user to present a full set of credentials to access sensitive information.

Building an IP-aware remember me service

One way to make remember me more secure would be to tie the user's IP address in to the cookie content. Let's walk through an example of how we would build our own RememberMeServices implementation to do just this.

The basic approach for this implementation is to extend the o.s.s.web.authentication.rememberme.TokenBasedRememberMeServices base class and extend it to allow for the addition of the requestor's IP address to both the cookie itself, and to the MD5 hash of the other remember me factors.

Extending the base class will involve overriding two key methods, and overriding or implementing some very minor helper methods. One other twist is that we'll have to temporarily store the HttpServletRequest (which we use to get the user's IP address) into a ThreadLocal, as some of the base class methods don't take HttpServletRequest as a parameter.

Extending TokenBasedRememberMeServices

First, we'll extend the `TokenBasedRememberMeServices` class and override specific behavior of the parent. Although the parent is override-friendly, there are some key bits of program flow that we'd prefer not to duplicate, so we'll make this class concise at the expense of being a little bit odd. Create the class in the `com.packtpub.springsecurity.security` package:

```
public class IPTokenBasedRememberMeServices extends
    TokenBasedRememberMeServices {
```

Some simple utility methods are used to set and get a `ThreadLocal` `HttpServletRequest`:

```
private static final ThreadLocal<HttpServletRequest> requestHolder =
new ThreadLocal<HttpServletRequest>();

public HttpServletRequest getContext() {
    return requestHolder.get();
}
public void setContext(HttpServletRequest context) {
    requestHolder.set(context);
}
```

We'll also add a utility method to get the IP address from the `HttpServletRequest`:

```
protected String getUserIPAddress(HttpServletRequest request) {
    return request.getRemoteAddr();
}
```

The first interesting method that we'll override is `onLoginSuccess`, which is used to set the cookie value for the remember me processor. In this method, we simply need to set the `ThreadLocal` and clear it after we're done. Keep in mind the parent method's flow—to aggregate all the information about the user's authenticated request and synthesize that into a cookie.

```
@Override
public void onLoginSuccess(HttpServletRequest request,
    HttpServletResponse response,
    Authentication successfulAuthentication) {
  try
  {
    setContext(request);
    super.onLoginSuccess(request, response, successfulAuthentication);
  }
  finally
  {
```

```
        setContext(null);
    }
}
```

The `onLoginSuccess` method of the parent class invokes `makeTokenSignature`
that is used to create the MD5 hash of the authentication credentials. We'll override
this to take the IP address from the request, and encode the returned cookie using a
utility class from the Spring framework:

```
@Override
protected String makeTokenSignature(long tokenExpiryTime,
    String username, String password) {
    return DigestUtils.md5DigestAsHex((username + ":" +
tokenExpiryTime + ":" + password + ":" + getKey() + ":" + getUserIPAdd
ress(getContext())).getBytes());
}
```

Similarly, we'll override the `setCookie` method to add an additional encoded bit that
includes the requesting IP address:

```
@Override
protected void setCookie(String[] tokens, int maxAge,
  HttpServletRequest request, HttpServletResponse response) {
  // append the IP adddress to the cookie
  String[] tokensWithIPAddress =
      Arrays.copyOf(tokens, tokens.length+1);
  tokensWithIPAddress[tokensWithIPAddress.length-1] =
      getUserIPAddress(request);
  super.setCookie(tokensWithIPAddress, maxAge,
      request, response);
}
```

This gives us all the pieces we need to set up the new cookie!

Finally, we'll override the `processAutoLoginCookie` method, which is used
to validate the contents of the remember me cookie that the user provided. The
superclass does most of the interesting work for us; however, due to the verbosity
of the parent method, we've made the choice to check the IP address before calling
the parent.

```
@Override
protected UserDetails processAutoLoginCookie(
  String[] cookieTokens,
  HttpServletRequest request, HttpServletResponse response)
{
  try
```

```
    {
       setContext(request);
    // take off the last token
       String ipAddressToken =
         cookieTokens[cookieTokens.length-1];
       if(!getUserIPAddress(request).equals(ipAddressToken))
       {
             throw new InvalidCookieException("Cookie IP Address did not
    contain a matching IP (contained '" + ipAddressToken + "')");
       }

       return super.processAutoLoginCookie(Arrays.copyOf(cookieTokens,
    cookieTokens.length-1), request, response);
    }
    finally
    {
       setContext(null);
    }
  }
}
```

Our code for the custom `RememberMeServices` is complete! Now we'll have to perform some minor configuration. Note that the full source code for this class (with additional comments) is included with the source for this chapter.

Configuring the custom RememberMeServices

Configuring the custom `RememberMeServices` implementation requires two steps. The first is to modify the `dogstore-base.xml` Spring configuration file to add a new Spring Bean definition for the class we just completed:

```
<bean class="com.packtpub.springsecurity.security.
IPTokenBasedRememberMeServices" id="ipTokenBasedRememberMeServicesBea
n">
  <property name="key"><value>jbcpPetStore</value></property>
  <property name="userDetailsService" ref="userService"/>
</bean>
```

The second minor set of changes is in the Spring Security XML configuration file. Modify the `<remember-me>` element to reference our custom Spring Bean as follows:

```
<remember-me key="jbcpPetStore"
   services-ref="ipTokenBasedRememberMeServicesBean"/>
```

Finally, add an `id` attribute to the `<user-service>` declaration, if it's not already there:

```
<user-service
   id="userService">
```

Restart the web application, and you should see the new IP filtering functionality take effect!

As the remember me cookie is Base64 encoded, we can verify our new addition by retrieving the value of the cookie and decoding it using a Base64 decoder tool. When we do this, we see a cookie named SPRING_SECURITY_REMEMBER_ME_COOKIE with content similar to the following:

```
guest:1251695034322:776f8ad44034f77d13218a5c431b7b34:127.0.0.1
```

You can see the IP address right at the end of the cookie, as we had expected. You'll also see the username, timestamp, and MD5 hash, respectively, prior to the newly added IP address.

> **Debugging remember me cookies**
>
> There are two difficulties when attempting to debug issues with remember me cookies. The first is getting the cookie value at all! Spring Security doesn't offer any log level that will log the cookie value that was set. We'd suggest a browser-based tool such as Chris Pederick's Web Developer plug-in (http://chrispederick.com/work/web-developer/) for Mozilla Firefox. Browser-based development tools typically allow selective examination (and even editing) of cookie values. The second (admittedly minor) difficulty is decoding the cookie value. You can feed the cookie value into an online or offline Base64 decoder (remember to add a trailing = sign to make it a valid Base64-encoded string!)

Note that IP-based remember me tokens may behave unexpectedly if the user is behind a shared or load balanced network infrastructure, such as a multi-WAN corporate environment. In most scenarios, however, the addition of an IP address to the remember me function provides an additional, welcome layer of security to a helpful user feature.

Customizing the remember me signature

Curious users may wonder if the expected value of the remember me form field checkbox, _spring_security_remember_me, or the cookie name, SPRING_SECURITY_REMEMBER_ME_COOKIE, can be changed, to obscure the use of Spring Security. While the <remember-me> declaration does not allow this flexibility, now that we've declared our own RememberMeServices implementation as a Spring Bean, we can simply define more properties to change the checkbox name and cookie name:

```
<bean class="com.packtpub.springsecurity.web.custom.
IPTokenBasedRememberMeServices" id="ipTokenBasedRememberMeServicesBea
n">
```

```
        <property name="key"><value>jbcpPetStore</value></property>
        <property name="userDetailsService" ref="userService"/>
        <property name="parameter" value="_remember_me"/>
        <property name="cookieName" value="REMEMBER_ME"/>
    </bean>
```

Don't forget to change the `login.jsp` page to set the name of the checkbox form field to match the `parameter` value we declared. We'd encourage you to do some experimentation here to ensure you understand how these settings are related.

Implementing password change management

We'll now walk through a basic extension to the in-memory `UserDetailsService` that allows a user to change their password. While this feature will be more useful when usernames and passwords are stored in the database, implementing it with an extension to the `o.s.s.core.userdetails.memory.InMemoryDaoImpl` will allow us to focus not on the storage mechanism, but the overall flow and design of this type of extension to the framework. In Chapter 4, we'll be extending our baseline further by moving to a database-backed credential store.

Extending the in-memory credential store to support password change

The `InMemoryDaoImpl` in-memory credential store supplied with the Spring Security framework uses a simple map to store usernames and their associated `UserDetails`. The `UserDetails` implementation utilized by `InMemoryDaoImpl` is `o.s.s.core.userdetails.User`, an implementation that we'll see in several other places in the Spring Security API.

The design of this extension is purposely simple and elides some important details, such as requiring the user to supply their old password prior to changing it. Adding these types of features is left as an exercise for the reader.

Extending InMemoryDaoImpl with InMemoryChangePasswordDaoImpl

We'll first write our own custom class to extend the baseline `InMemoryDaoImpl`, and add a method to allow us to change the user's password. As `User` is an immutable object, we'll have to essentially copy the existing `User` object, but replace the password with the one that the user provided. Let's declare an interface that we'll reuse in later chapters, which illustrates a method which provides password change functionality:

```
package com.packtpub.springsecurity.security;
// imports omitted
public interface IChangePassword extends UserDetailsService {  void
changePassword(String username, String password);
}
```

The following code provides the change password functionality for the in-memory user data store:

```
package com.packtpub.springsecurity.security;

public class InMemoryChangePasswordDaoImpl extends InMemoryDaoImpl
implements IChangePassword {
  @Override
  public void changePassword(String username,
                String password) {
    // get the UserDetails
    User userDetails =
        (User) getUserMap().getUser(username);
    // create a new UserDetails with the new password
    User newUserDetails =
        new User(userDetails.getUsername(),password,
        userDetails.isEnabled(),
        userDetails.isAccountNonExpired(),
      userDetails.isCredentialsNonExpired(),
        userDetails.isAccountNonLocked(),
        userDetails.getAuthorities());
    // add to the map
    getUserMap().addUser(newUserDetails);
  }
}
```

Fortunately, few tricks are required to add this simple bit of functionality to our custom subclass. Let's review the configuration requirements needed to add this custom `UserDetailsService` to our pet store.

Configuring Spring Security to use InMemoryChangePasswordDaoImpl

Now, we'll need to reconfigure the Spring Security XML configuration file to use our new `UserDetailsService` implementation. Unfortunately, this is a bit harder than we'd hope, as the `<user-service>` element is given special treatment in the Spring Security configuration processor. Instead, we'll have to explicitly declare our own bean and remove the `<user-service>` element we had previously declared. We'll change this:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider>
    <user-service id="userService">
      <user authorities="ROLE_USER" name="guest" password="guest"/>
    </user-service>
  </authentication-provider>
</authentication-manager>
```

to this:

```
<authentication-provider user-service-ref="userService"/>
```

The `user-service-ref` attribute, as we expect by now, is a reference to a Spring Bean with the `id` of userService. Thus, in our `dogstore-base.xml` Spring Beans XML configuration file, we declare the following bean:

```
<bean id="userService" class="com.packtpub.springsecurity.security.
InMemoryChangePasswordDaoImpl">
  <property name="userProperties">
    <props>
      <prop key="guest">guest,ROLE_USER</prop>
    </props>
  </property>
</bean>
```

You'll note that the syntax for declaration of users isn't as readable as the `<user>` elements contained within the `<user-service>` declaration. Unfortunately, the `<user>` elements are available only when using the default `InMemoryDaoImpl` implementation, and can't be reused with a custom `UserDetailsService`. For the purposes of our example, this restriction makes things slightly more complex, but in reality, storing user definitions in the configuration file isn't something anyone's likely to do for long! For those who are interested, *Section 6.2* of the Spring Security 3 reference document describes the full details of the comma-separated declarative syntax for supplying user information.

> **Making effective use of an in-memory UserDetailsService**
>
> A very common scenario for the use of an in-memory `UserDetailsService` and hard-coded user lists is the authoring of unit tests for secured components. Unit test authors often code or configure the minimal context to test the functionality of the component under test. Using an in-memory `UserDetailsService` with a well-defined set of users and `GrantedAuthority` values provides the test author with an easily controlled test environment.

At this point, you should be able to start the JBCP Pets application without any reports of configuration errors. We'll finish wiring the change password functionality up to the UI in the final two steps of this exercise.

Building a change password page

We'll build a simple password change page that will allow the user to change their password.

This page will be tied into the **My Account** page through a simple link. First, we'll add the link to `/account/home.jsp` file:

```
<p>
   Please find account functions below...
</p>
<ul>
   <li><a href="changePassword.do">Change Password</a></li>
</ul>
```

Next, we'll build the **Change Password** page itself in `/account/changePassword.jsp`:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
   pageEncoding="ISO-8859-1"%>
```

```
<jsp:include page="../common/header.jsp">
  <jsp:param name="pageTitle" value="Change Password"/>
</jsp:include>
<h1>Change Password</h1>
<form method="post">
  <label for="password">New Password</label>:
  <input id="password" name="password" size="20" maxlength="50"
type="password"/>
  <br />
  <input type="submit" value="Change Password"/>
</form>
<jsp:include page="../common/footer.jsp"/>
```

Finally, we'll need to augment our Spring MVC-based `AccountController` to handle the submission of the password change request (we've not covered `AccountController` in prior chapters, but it has a simple handler for the account home page).

Adding a change password handler to AccountController

We'll need to add an injected reference to our custom `UserDetailsService` to the `com.packtpub.springsecurity.web.controller.AccountController`, so that we can make use of its change password functionality. Spring's `@Autowired` annotation makes this trivial:

```
@Autowired
private IChangePassword changePasswordDao;
```

Two request handler methods will take care of rendering the form, and processing the form submission via `POST`:

```
@RequestMapping(value="/account/changePassword.
do",method=RequestMethod.GET)
public void showChangePasswordPage() {
}
@RequestMapping(value="/account/changePassword.
do",method=RequestMethod.POST)
public String submitChangePasswordPage(@RequestParam("password")
String newPassword) {
  Object principal = SecurityContextHolder.getContext().
getAuthentication().getPrincipal();
  String username = principal.toString();
  if (principal instanceof UserDetails) {
    username = ((UserDetails)principal).getUsername();
```

```
    }
    changePasswordDao.changePassword(username, newPassword);
    SecurityContextHolder.clearContext();
    return "redirect:home.do";
}
```

Once you complete these changes, restart the application and look for the **Change Password** action under the **My Account** section of the site.

Exercise notes

The astute reader will observe that this change password form is far too simple for real-world use. Indeed, many password change implementations will be much more complex, and will include features such as:

- Password confirmation—making sure the user types the new password correctly in two separate boxes.
- Old password confirmation—adding a layer of security by requiring that the users supply their old password in order to change it (this is especially important when the remember me feature is in use).
- Password rule validity checking—checking for password complexity and weak or insecure passwords.

You may also have noticed when you tried the feature that you were automatically logged out. This was caused by the `SecurityContextHolder.clearContext()` call, which effectively removes the user's `SecurityContext` from the request, and requires them to authenticate again. In practice, we would want to warn the user that this would occur, or find workarounds that don't require the user to authenticate again.

Summary

This chapter allowed us to explore the authenticated user's lifecycle in more detail, and experiment with structural modifications to the JBCP Pets store. We've taken one step closer for satisfying the security auditors by adding true login and logout functionality, and made users happier by unifying the site experience. We also learned the following skills:

- Configuration and use of a Spring MVC-based custom login page
- Configuration of Spring Security logout functionality
- Enabling of the remember me feature
- Customization of the remember me feature to track user IP address
- Implementation of change password functionality
- Customization of the `UserDetailsService` and `InMemoryDaoImpl`

In Chapter 4, we'll make the exciting leap to a database-backed authentication store, and learn how to ensure the security of passwords and other sensitive data stored in databases.

4
Securing Credential Storage

Up to this point, we've updated the JBCP Pets site with user-friendly functionality, including a custom login page, and change password and remember me features.

In this chapter, we'll make a leap to a database-backed authentication store from the in-memory store we have used in the book until this point. We'll explore the default expected Spring Security database schema, and will look into ways to extend JDBC implementation with customization.

During the course of this chapter, we'll:

- Understand how to configure Spring Security to utilize the services of a JDBC-accessible database to store and authenticate users
- Learn how to configure a JDBC-compatible in-memory database using HSQLDB, for developer testing purposes
- Work through the process of adapting Spring Security JDBC to an existing legacy schema
- Examine two techniques for user and password management, both with out of the box and custom techniques
- Examine different methods of configuring password encoding
- Understand the password salting technique of providing additional security to stored passwords
- Solve the problem of allowing user remember me tokens to persist even if the server restarts
- Secure the application transport layer by understanding how to configure SSL/TLS encryption and port mapping

Database-backed authentication with Spring Security

An obvious issue with our more security-conscious implementation of JBCP Pets is that our in-memory storage of users and passwords is too short-lived to be user-friendly. As soon as the application is restarted, any new user registrations, password changes, or other activity will be lost. This isn't acceptable, so the next logical implementation step when securing JBCP Pets will be to reconfigure Spring Security to utilize a relational database for user storage and authentication. The use of a JDBC-accessible relational database will allow user data to persist through application server restarts, and is more typical of real-world Spring Security use.

Configuring a database-resident authentication store

The first portion of this exercise involves setting up an instance of the Java-based relational database HyperSQL DB (or HSQL, for short), populated with the Spring Security default schema. We'll configure HSQL to run in-memory using Spring 3's embedded database configuration feature—a significantly simpler method of configuration than setting up the database by hand.

Keep in mind that in this example (and the remainder of the book), we'll use HSQL, primarily, due to its ease of setup. We encourage you to tweak the configuration and use the database of your preference if you're following along with the examples. As we didn't want this portion of the book to focus on the complexities of database setup, we chose convenience over realism for the purposes of the exercises.

Creating the default Spring Security schema

We've supplied an SQL file, `security-schema.sql`, which will create all the tables required to implement Spring Security using HSQL. If you're following along with your own database instance, you may have to adjust the schema definition syntax to fit your particular database. We'll place this SQL file so that it's on the classpath in `WEB-INF/classes`.

Configuring the HSQL embedded database

To configure the HSQL embedded database, we'll modify the `dogstore-security.xml` file to both set up the database and run SQL to create the Spring Security table structure. First, we'll add a reference to the `jdbc` XML schema definition at the top of the file:

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xsi:schemaLocation="
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/
        spring-security3.0.xsd"
>
```

Next, we'll declare the `<embedded-database>` element, along with a reference to the SQL script:

```
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:security-schema.sql"/>
</jdbc:embedded-database>
```

If you start the server at this point, you should be able to see the initialization of the HSQL database in the logs. Remember that the `<embedded-database>` declaration creates this database only in the memory, so you won't see anything on disk, and you won't be able to use standard tools to query it.

Configuring JdbcDaoImpl authentication store

We'll modify the `dogstore-security.xml` file to declare that we're using a JDBC `UserDetailsService` implementation, instead of the Spring Security in-memory `UserDetailsService` that we configured in *Chapter 2*, *Getting Started with Spring Security* and *Chapter 3*, *Enhancing the User Experience*. This is done with a simple change to the `<authentication-manager>` declaration:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider>
    <jdbc-user-service data-source-ref="dataSource"/>
  </authentication-provider>
</authentication-manager>
```

> The `data-source-ref` refers to the bean we declared using the shortcut `<embedded-database>` declaration in the previous step.

Adding user definitions to the schema

Finally, we'll create another SQL file that will get executed when the in-memory database is created. This SQL file will contain information about our default users, `admin` and `guest`, with the same `GrantedAuthority` settings that we've used in prior chapters. We'll call this SQL file `test-data.sql`, and we'll put it alongside `security-schema.sql` in `WEB-INF/classes`:

```
insert into users(username, password, enabled) values
  ('admin','admin',true);
insert into authorities(username,authority) values
  ('admin','ROLE_USER');
insert into authorities(username,authority) values
  ('admin','ROLE_ADMIN');
insert into users(username, password, enabled) values
  ('guest','guest',true);
insert into authorities(username,authority) values
  ('guest','ROLE_USER');
commit;
```

Next, we'll need to add this SQL file to the embedded database configuration so that it is loaded at startup:

```
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:security-schema.sql"/>
  <jdbc:script location="classpath:test-data.sql"/>
</jdbc:embedded-database>
```

After the SQL is added to the embedded database configuration, we should be able to start the application and log in. Spring Security is now looking at the database for authentication and `GrantedAuthority` information!

How database-backed authentication works

You may recall from our examination of the authentication process in Chapter 2 that the `AuthenticationManager` delegates to `AuthenticationProvider` to validate the credentials of the principal and ensure that it should be able to access the system at all. The `AuthenticationProvider` that we have been using in Chapters 2 and 3 was the `DaoAuthenticationProvider`. This provider delegates to a `UserDetailsService` implementation to retrieve and validate the information about the principal from the credential store. We can see this in the diagram from Chapter 2:
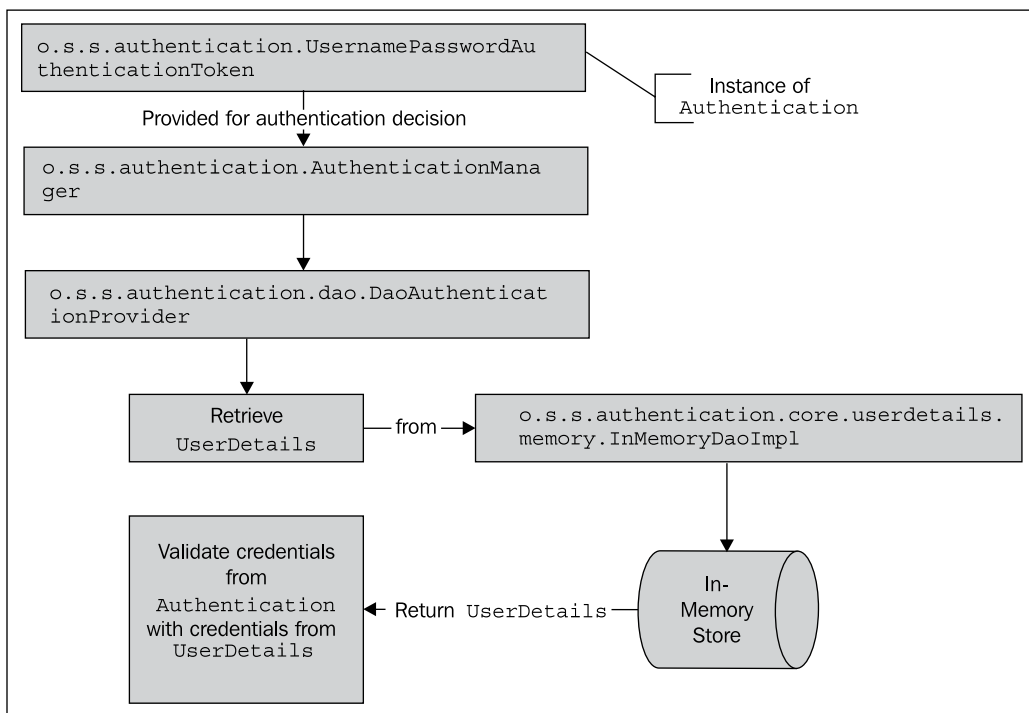
As you may anticipate, the only meaningful difference between our configuration of a database-backed authentication store and the in-memory store is the implementation of the `UserDetailsService`. The `o.s.s.core.userdetails.jdbc.JdbcDaoImpl` class provides an implementation of a `UserDetailsService`. Instead of looking at an in-memory store (populated from the Spring Security XML configuration), the `JdbcDaoImpl` looks up users in a database.
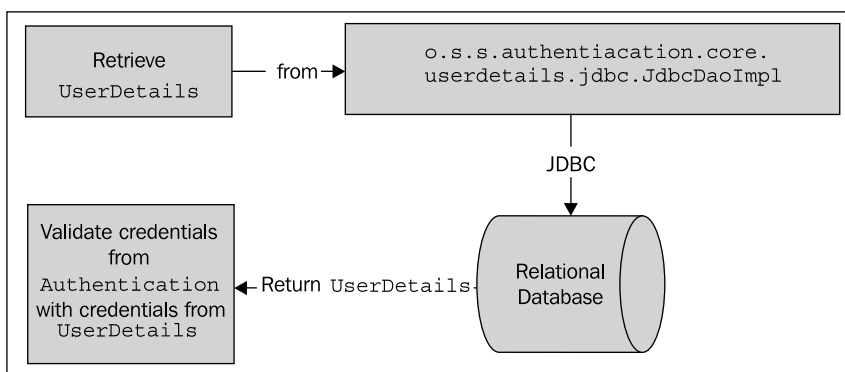
You may note that we didn't reference the implementation class at all. This is because the `<jdbc-user-service>` declaration in the updated Spring Security configuration will automatically configure the `JdbcDaoImpl` and wire it up to the `AuthenticationProvider`. Later in this chapter, we'll see how to configure Spring Security to use our own implementation of `JdbcDaoImpl`, which continues to support the change password feature that we added to our custom `InMemoryDaoImpl` in Chapter 3. Let's examine the configuration required to implement our own `JdbcDaoImpl` subclass that supports the change password function.

Implementing a custom JDBC UserDetailsService

As we did in one of the exercises in the previous chapter, we'll take the baseline `JdbcDaoImpl` as our starting point, and extend it to support a change password function.

Creating a custom JDBC UserDetailsService class

Create the following class in the `com.packtpub.springsecurity.security` package:

```
public class CustomJdbcDaoImpl extends JdbcDaoImpl implements
IChangePassword {
  public void changePassword(String username, String password) {
    getJdbcTemplate()
    update("UPDATE USERS SET PASSWORD = ? WHERE USERNAME = ?",
           password, username);
  }
}
```

You can see that this simple class extends the default `JdbcDaoImpl` with a function to update the password in the database to the new password that the user ostensibly requested. We use standard Spring JDBC functionality to do this.

Adding a Spring Bean declaration for the custom UserDetailsService

Add the following Spring Bean declaration to the `dogstore-base.xml` Spring configuration file:

```
<bean id="jdbcUserService"
      class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

Again, the `dataSource` Bean reference here resolves to the `<embedded-database>` declaration we made to set up the HSQL in-memory database.

You'll observe that the custom `UserDetailsService` implementation allows us to tweak the interaction with the database significantly. We'll use this capability to expand the baseline functionality of the `UserDetailsService` in later examples. This type of customization is very common in complex applications of Spring Security.

Out of the box JDBC-based user management

As our simple extension to `JdbcDaoImpl` illustrated, one might extend the class, while retaining the baseline functionality at the same time. But what if we wanted to implement more advanced features, such as user registration (a must for an online store!) and user management features, allowing site administrators to create users, update passwords, and so on?

Although these types of functions are relatively easy to write with additional JDBC statements, Spring Security actually provides out of the box functionality to support many common **Create**, **Read**, **Update**, and **Delete** (**CRUD**) operations on users in JDBC databases. This can be convenient for simple systems, and a good base to build on for any custom requirements that a user may have.

The implementation class `o.s.s.provisioning.JdbcUserDetailsManager` conveniently extends `JdbcDaoImpl` for us, and provides a number of helpful user-related methods, declared as part of the `o.s.s.provisioning.` `UserDetailsManager` interface:

Method	Description
`void createUser(UserDetails user)`	Creates a new user with the given `UserDetails`, including any declared `GrantedAuthority`.
`void updateUser(final UserDetails user)`	Updates a user with the given `UserDetails`. Updates `GrantedAuthority` and removes the user from the user cache.
`void deleteUser(String username)`	Deletes the user with the given username, and removes the user from the user cache.
`boolean userExists(String username)`	Indicates whether or not a user (whether active or inactive) exists with the given username.
`void changePassword(String oldPassword, String newPassword)`	Changes the password of the currently logged-in user. The user must supply the correct current password in order for the operation to succeed.

As you can see, the `changePassword` method on `JdbcUserDetailsManager` precisely fits a gap in the functionality of our `CustomJdbcDaoImpl` class – it will verify the user's existing password when changing it. Let's review the configuration steps required to replace our `CustomJdbcDaoImpl` with the `JdbcUserDetailsManager`.

First, we'll have to make need to declare the `JdbcUserDetailsManager` bean in `dogstore-base.xml`:

```
<bean id="jdbcUserService"
      class="org.springframework.security
             .provisioning.JdbcUserDetailsManager">
   <property name="dataSource" ref="dataSource"/>
   <property name="authenticationManager"
             ref="authenticationManager"/>
</bean>
```

The reference to `AuthenticationManager` matches the `alias` we've previously declared in the `<authentication-manager>` element in `dogstore-security.xml`. Don't forget to comment out the declaration of the `CustomJdbcDaoImpl` bean—we will (temporarily) not be using it.

Next, we'll have to make some minor adjustments to the `changePassword.jsp` page:

```
<h1>Change Password</h1>
<form method="post">
  <label for="oldpassword">Old Password</label>:
  <input id="oldpassword" name="oldpassword"
         size="20" maxlength="50" type="password"/>
  <br />
  <label for="password">New Password</label>:
  <input id="password" name="password" size="20"
         maxlength="50" type="password"/>
  <br />
```

Finally, we'll have to make some minor adjustments to `AccountController`. Replace the `@Autowired` reference to the `IChangePassword` implementation with:

```
@Autowired
private UserDetailsManager userDetailsManager;
```

The `submitChangePasswordPage` also becomes much simpler, as we are relying on information about the current authenticated principal which the `JdbcUserDetailsManager` determines on our behalf:

```
public String submitChangePasswordPage(@RequestParam("oldpassword")
        String oldPassword,
   @RequestParam("password") String newPassword) {
```

```
    userDetailsManager.changePassword(oldPassword, newPassword);
    SecurityContextHolder.clearContext();
    return "redirect:home.do";
}
```

Once these changes are complete, you can restart the web application and try out the new change password functionality!

Notice what happens when you don't supply the correct password. Why do you think this happens? Try to think through how you'd adjust the behavior to make it more user friendly.

Although we won't demonstrate all of the functionality supported by `JdbcUserDetailsManager`, we can see that it would be very easy to wire simple JSP pages (properly secured with authorization, of course!) to allow administrators to manage users of the site—essential for a production website!

Advanced configuration of JdbcDaoImpl

`JdbcDaoImpl` has a number of configuration options that allow for adapting its use to an existing schema, or for more sophisticated adjustment of its existing capabilities. In many cases, it's possible to adapt the configuration of the out of the box JDBC `UserDetailsService` without having to write your own code.

One important feature is the ability to add a level of indirection between users and `GrantedAuthority` declarations by grouping `GrantedAuthority` into logical sets called **groups**. Users are then assigned one or more groups, whose membership confers a set of `GrantedAuthority` declarations.
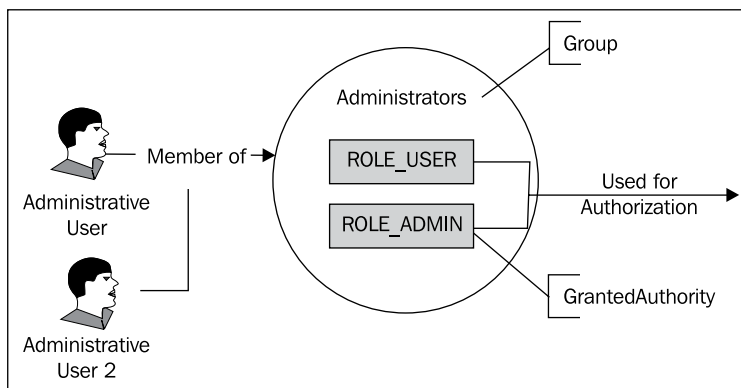
As you see in the diagram, this indirection allows the assignment of the same set of roles to multiple users, by simply assigning any new users to existing groups. Contrast this with the behavior we've seen this far, where we assigned `GrantedAuthority` directly to individual users.

This bundling of common sets of authorities can be helpful in the following scenarios:

- You need to segregate users into communities, with some overlapping roles between groups.

- You want to globally change authorization for a class of user. For example, if you have a "supplier" group, you might want to enable or disable their access to particular portions of the application.

- You have a large number of users, and you don't need a user-level authority configuration.

Unless your application has a very small user base, there is a very high likelihood that you'll be using group-based access control. The management ease and flexibility of this rights management approach far outweighs the slightly greater complexity. This indirect technique of aggregating user privileges by group is commonly referred to as **Group-Based Access Control** (**GBAC**).

> Group-based access control is an approach common to almost every secured operating system or software package in the market. Microsoft **Active Directory** (**AD**) is one of the most visible implementations of large-scale GBAC, due to its design of slotting AD users into groups and assignment of privileges to those groups. Management of privileges in large AD-based organizations is made exponentially simpler through the use of GBAC.
>
> Try to think of the security models of the software you use—how are users, groups, and privileges managed? What are the pros and cons of the way the security model is written?

Let's add a level of abstraction to JBCP Pets and apply the concept of group-based authorization to the site.

Configuring group-based authorization

We'll add two groups to the website—regular users, which we'll call "Users", and administrative users, which we'll call "Administrators". Our existing `guest` and `admin` user accounts will be dropped into the appropriate groups through modifications to our SQL script that we use to set up the database.

Configuring JdbcDaoImpl to use groups

First, we must set properties on our `JdbcDaoImpl` custom subclass to enable the use of groups, and disable the use of direct authority granting to users. Add the following to the bean definition in `dogstore-base.xml`:

```
<bean id="jdbcUserService"
      class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
  <property name="dataSource" ref="dataSource"/>
  <property name="enableGroups" value="true"/>
  <property name="enableAuthorities" value="false"/>
</bean>
```

Note that if you are following along and still have the code and configuration changes in place to utilize the `JdbcUserManager`, please revert them as we'll be using the `CustomJdbcDaoImpl` for the remainder of this chapter.

Modifying the initial load SQL script

We'll simply modify the SQL that we use to populate the database to:

- Define our groups
- Assign `GrantedAuthority` specifications to groups
- Assign users to groups

For simplicity, we'll create a new SQL script called `test-users-groups-data.sql`.

We'll add the groups first:

```
insert into groups(group_name) values ('Users');
insert into groups(group_name) values ('Administrators');
```

Next, assign roles to groups:

```
insert into group_authorities(group_id, authority) select id,'ROLE_
USER' from groups where group_name='Users';
insert into group_authorities(group_id, authority) select id,'ROLE_
USER' from groups where group_name='Administrators';
insert into group_authorities(group_id, authority) select id,'ROLE_
ADMIN' from groups where group_name='Administrators';
```

Next, create the users:

```
insert into users(username, password, enabled) values
('admin','admin',true);
insert into users(username, password, enabled) values
('guest','guest',true);
```

Finally, assign users to groups:

```
insert into group_members(group_id, username) select id,'guest' from
groups where group_name='Users';
insert into group_members(group_id, username) select id,'admin' from
groups where group_name='Administrators';
```

Modifying the embedded database creation declaration

We'll need to update the creation of our embedded HSQL database to reference this script in lieu of the existing `test-data.sql` script:

```
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:security-schema.sql"/>
  <jdbc:script location="classpath:test-users-groups-data.sql"/>
</jdbc:embedded-database>
```

Note that the `security-schema.sql` already contains declarations for the tables required to support the group structure, so we don't need to modify that script.

At this point, you should be able to start the JBCP Pets site again and it will behave just as before; however, the additional layer of abstraction between users and privileges will enable us to develop some sophisticated user management functionality in later exercises with a fraction of effort.

Let's step away from the JBCP Pets scenario for a moment and cover one more important bit of configuration while we're in this area.

Using a legacy or custom schema with database-resident authentication

It's common for new users of Spring Security to begin their experience by adapting the JDBC user, group, or role mapping to an existing schema. Even though a legacy database doesn't conform to the expected Spring Security schema, we can still configure the `JdbcDaoImpl` to map to it.

Imagine we have an existing legacy database schema similar to the following figure, onto which we are going to implement Spring Security.
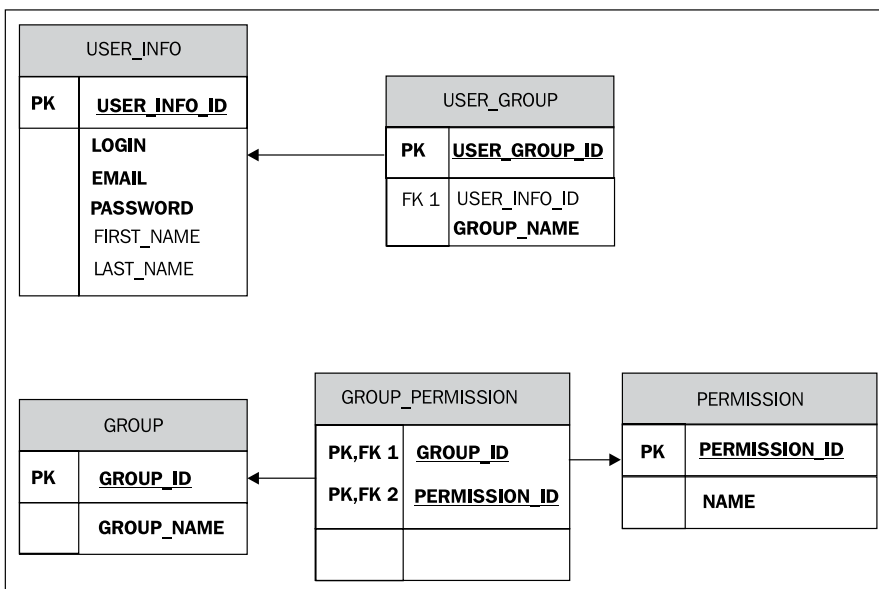
We can easily change the configuration of `JdbcDaoImpl` to utilize this schema and override the Spring Security expected table definitions and columns that we're using for JBCP Pets.

Determining the correct JDBC SQL queries

`JdbcDaoImpl` has three SQL queries which have a well-defined parameter and set of returned columns. We must determine the SQL that we'll assign to each of these queries, based on its intended functionality. Each SQL query used by the `JdbcDaoImpl` takes the username presented at login as its one and only parameter.

Query name	Description	Expected SQL columns
usersByUsernameQuery	Returns one or more users matching the username. Only the first user is used.	**Username** (string) **Password** (string) **Enabled** (Boolean)
authoritiesByUsernameQuery	Returns one or more granted authorities directly provided to the user. Typically used when GBAC is disabled.	**Username** (string) **Granted Authority** (string)

Query name	Description	Expected SQL columns
`groupAuthoritiesByUsernameQuery`	Returns granted authorities and group details provided to the user through group membership. Used when GBAC is enabled.	**Group Primary Key** (any) **Group Name** (any) **Granted Authority** (string)

Be aware that in some cases, the return columns are not used by the default `JdbcDaoImpl` implementation, but they must be returned anyway. Spend some time now trying to write these queries for the database diagram on the previous page before moving on to the next step.

Configuring the JdbcDaoImpl to use custom SQL queries

In order to use the custom SQL queries for our non-standard schema, we'll simply configure the `JdbcDaoImpl` properties in the Spring Bean configuration file. Note that in order to configure the JDBC queries on `JdbcDaoImpl`, you cannot use the `<jdbc-user-service>` declaration. You must explicitly instantiate the bean, as we've done with our custom `JdbcDaoImpl`.

```
<bean id="jdbcUserService"
      class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
  <property name="dataSource" ref="dataSource"/>
  <property name="enableGroups" value="true"/>
  <property name="enableAuthorities" value="false"/>
  <property name="usersByUsernameQuery">
    <value>SELECT LOGIN, PASSWORD,
          1 FROM USER_INFO WHERE LOGIN = ?
    </value>
  </property>
  <property name="groupAuthoritiesByUsernameQuery">
    <value>SELECT G.GROUP_ID, G.GROUP_NAME, P.NAME
      FROM USER_INFO U
      JOIN USER_GROUP UG on U.USER_INFO_ID = UG.USER_INFO_ID
      JOIN GROUP G ON UG.GROUP_ID = G.GROUP_ID
      JOIN GROUP_PERMISSION GP ON G.GROUP_ID = GP.GROUP_ID
      JOIN PERMISSION P ON GP.PERMISSION_ID = P.PERMISSION_ID
      WHERE U.LOGIN = ?
    </value>
  </property>
</bean>
```

This is the only configuration required to use Spring Security to read settings from an existing, non-default schema! Keep in mind that utilization of an existing schema commonly requires extension of the `JdbcDaoImpl` to support changing of passwords, renaming of user accounts, and other user-management functions.

If you are using the `JdbcUserDetailsManager` to perform user management tasks, be aware that (as of this writing) only some of the over twenty SQL queries utilized by the class are accessible through configuration. Please refer to the Javadoc or source code to review the defaults for the queries used by the `JdbcUserDetailsManager`.

Configuring secure passwords

We recall from the security audit in *Chapter 1*, *Anatomy of an Unsafe Application* that the security of passwords stored in **cleartext** was a top priority of the auditors. In fact, in any secured system, password security is a critical aspect of trust and authoritativeness of an authenticated principal. Designers of a fully secured system must ensure that passwords are stored in a way in which malicious users would have an impractically difficult time compromising them.

The following general rules should be applied to passwords stored in a database:

- Passwords must not be stored in cleartext (plain text)
- Passwords supplied by the user must be compared to recorded passwords in the database
- A user's password should not be supplied to the user upon demand (even if the user forgets it)

For the purposes of most applications, the best fit for these requirements involves one-way encoding or encryption of passwords as well as some type of randomization of the encrypted passwords. One-way encoding provides the security and uniqueness properties that are important to properly authenticate users with the added bonus that once encrypted, the password cannot be decrypted.

In most secure application designs, it is neither required nor desirable to ever retrieve the user's actual password upon request, as providing the user's password to them without proper additional credentials could present a major security risk. Most applications instead provide the user the ability to reset their password, either by presenting additional credentials (such as their social security number, date of birth, tax ID, or other personal information), or through an email-based system.

Storing other types of sensitive information

Many of the guidelines listed that apply to passwords apply equally to other types of sensitive information, including social security numbers and credit card information (although, depending on the application, some of these may require the ability to decrypt).

It's quite common for databases storing this type of information to represent it in multiple ways, for example, a customer's full 16-digit credit card number would be stored in a highly encrypted form, but the last four digits might be stored in cleartext (for reference, think of any internet commerce site that displays **XXXX XXXX XXXX 1234** to help you identify your stored credit cards).

You may already be thinking ahead and wondering, given our (admittedly unrealistic) approach of using SQL to populate our HSQL database with users, how do we encode the passwords? HSQL, or most other databases for that matter, don't offer encryption methods as built-in database functions.

Typically, the **bootstrap** process (populating a system with initial users and data) is handled through some combination of SQL loads and Java code. Depending on the complexity of your application, this process can get very complicated.

For the JBCP Pets application, we'll retain the embedded-database declaration and the corresponding SQL, and then add a small bit of Java to fire after the initial load to encrypt all the passwords in the database. For password encryption to work properly, two actors must use password encryption in synchronization ensuring that the passwords are treated and validated consistently.
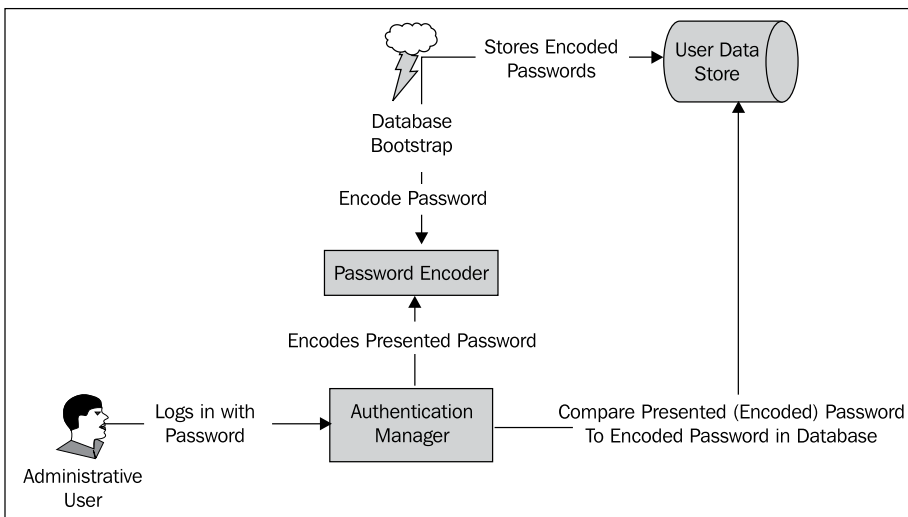
Password encryption in Spring Security is encapsulated and defined by implementations of the `o.s.s.authentication.encoding.PasswordEncoder` interface. Simple configuration of a password encoder is possible through the `<password-encoder>` declaration within the `<authentication-provider>` element as follows:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider user-service-ref="jdbcUserService">
    <password-encoder hash="sha"/>
  </authentication-provider>
</authentication-manager>
```

You'll be happy to learn that Spring Security ships with a number of implementations of `PasswordEncoder`, which are applicable for different needs and security requirements. The implementation used can be specified using the `hash` attribute of the `<password-encoder>` declaration.

The following table provides a list of the out of the box implementation classes and their benefits. Note that all implementations reside in the `o.s.s.authentication.encoding` package.

Implementation class	Description	hash value
PlaintextPasswordEncoder	Encodes the password as plaintext. Default `DaoAuthenticationProvider` password encoder.	plaintext
Md4PasswordEncoder	`PasswordEncoder` utilizing the MD4 hash algorithm. MD4 is not a secure algorithm—use of this encoder is not recommended.	md4
Md5PasswordEncoder	`PasswordEncoder` utilizing the MD5 one-way encoding algorithm.	md5
ShaPasswordEncoder	`PasswordEncoder` utilizing the SHA one-way encoding algorithm. This encoder can support configurable levels of encoding strength.	sha sha-256
LdapShaPasswordEncoder	Implementation of LDAP SHA and LDAP SSHA algorithms used in integration with LDAP authentication stores. We'll learn more about this algorithm in *Chapter 9, LDAP Directory Services* where we cover LDAP.	{sha} {ssha}

As with many other areas of Spring Security, it's also possible to reference a bean definition implementing `PasswordEncoder` to provide more precise configuration and allow the `PasswordEncoder` to be wired into other beans through dependency injection. For JBCP Pets, we'll need to use this bean reference method in order to encode the bootstrapped user data.

Let's walk through the process of configuring basic password encoding for the JBCP Pets application.

Configuring password encoding

Configuring basic password encoding involves two pieces—encrypting the passwords we load into the database after the SQL script executes, and ensuring that the `DaoAuthenticationProvider` is configured to work with a `PasswordEncoder`.

Configuring the PasswordEncoder

First, we'll declare an instance of a `PasswordEncoder` as a normal Spring bean:

```
<bean class="org.springframework.security.authentication.
            encoding.ShaPasswordEncoder" id="passwordEncoder"/>
```

You'll note that we're using the SHA-1 `PasswordEncoder` implementation. This is an efficient one-way encryption algorithm, commonly used for password storage.

Configuring the AuthenticationProvider

We'll need to configure the `DaoAuthenticationProvider` to have a reference to the `PasswordEncoder`, so that it can encode and compare the presented password during user login. Simply add a `<password-encoder>` declaration and refer to the bean ID we defined in the previous step:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider user-service-ref="jdbcUserService">
    <password-encoder ref="passwordEncoder"/>
  </authentication-provider>
</authentication-manager>
```

Try to start the application at this point, and then try to log in. You'll notice that what were previously valid login credentials are now being rejected. This is because the passwords stored in the database (loaded with the bootstrap `test-users-groups-data.sql` script) are not stored in an encrypted form that matches the password encoder. We'll need to post-process the bootstrap data with some simple Java code.

Writing the database bootstrap password encoder

The approach we'll take for encoding the passwords loaded via SQL is to have a Spring bean that executes an init method after the embedded-database bean is instantiated. The code for this bean, com.packtpub.springsecurity.security. DatabasePasswordSecurerBean, is fairly simple.

```
public class DatabasePasswordSecurerBean extends JdbcDaoSupport {
  @Autowired
  private PasswordEncoder passwordEncoder;

  public void secureDatabase() {
    getJdbcTemplate().query("select username, password from users",
                            new RowCallbackHandler(){
      @Override
      public void processRow(ResultSet rs) throws SQLException {
        String username = rs.getString(1);
        String password = rs.getString(2);
        String encodedPassword =
                passwordEncoder.encodePassword(password, null);
        getJdbcTemplate().update("update users set password = ?
          where username = ?", encodedPassword,username);
        logger.debug("Updating password for username:
                    "+username+" to: "+encodedPassword);
      }
    });
  }
}
```

The code uses the Spring JdbcTemplate functionality to loop through all the users in the database and encode the password using the injected PasswordEncoder reference. Each password is updated individually.

Configuring the bootstrap password encoder

We need to configure the Spring bean declaration such that the bean is initialized upon start of the web application and after the <embedded-database> bean. Spring bean dependency tracking ensures that the DatabasePasswordSecurerBean executes at the proper time:

```
<bean class="com.packtpub.springsecurity.security.
            DatabasePasswordSecurerBean"
            init-method="secureDatabase" depends-on="dataSource">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

If you start the JBCP Pets application at this point, you'll see that the passwords in the database are encoded, and login now functions properly.

Would you like some salt with that password?

If the security auditor were to examine the encoded passwords in the database, he'd find something that would still make him concerned about the website's security. Let's examine what the stored username and password values are for our `admin` and `guest` users:

Username	Plaintext password	Encrypted password
admin	admin	7b2e9f54cdff413fcde01f330af6896c3cd7e6cd
guest	guest	2ac15cab107096305d0274cd4eb86c74bb35a4b4

This looks very secure—the encrypted passwords obviously bear no resemblance to the original passwords. What could the auditor be concerned about? What if we add a new user who happens to have the same password as our `admin` user?

Username	Plaintext Password	Encrypted Password
fakeadmin	admin	7b2e9f54cdff413fcde01f330af6896c3cd7e6cd

Now, note that the encrypted password of the `fakeadmin` user is exactly the same as the real `admin` user! Thus a hacker who had somehow gained the ability to read the encrypted passwords in the database could compare their known password's encrypted representation with the unknown one for the admin account, and see they are the same! If the hacker had access to an automated tool to perform this analysis, they could likely compromise the administrator's account within a matter of hours.

> Having personally worked with a database where passwords were encrypted in exactly this way, my engineering team and I decided to run a little experiment and see what the SHA-1 encrypted value of the plaintext password `password` was. We then took the encrypted form of the word `password` and ran a database query to see how many users in the database had this highly insecure password. To our surprise and dismay, we found many, including a VP of the organization. Each user got a nice follow-up email reminder about the benefits of choosing a hard to guess password, and development quickly got to work on a more secure password encryption mechanism!

Recall our mention in Chapter 3 of rainbow table techniques that malicious users can use to determine user passwords if they have access to the database. These (and other) hacking techniques take advantage of the fact that hash algorithms are deterministic—the same input always leads to the same output, and as such, if an attacker tries enough inputs, they may happen to match an unknown output to the result of a known input.

One common, and effective, method of adding another layer of security to encrypted passwords is to incorporate a **salt**. A salt is a second plaintext component which is concatenated with the plaintext password prior to encryption in order to ensure that two factors must be used to generate (and thus compare) encrypted password values. Properly selected salts can guarantee that no two passwords will ever have the same encrypted value, thus preventing the scenario that concerned our auditor, and avoiding many common types of brute force password cracking techniques.

Best practice salts generally fall into one of two categories:

- They are algorithmically generated from some piece of data associated with the user—for example, the timestamp that the user was created
- They are randomly generated, and stored in some form (plaintext or two-way encrypted) along with the user's password record

For example, the following diagram illustrates the simple case where the salt is the same as the user's login name:
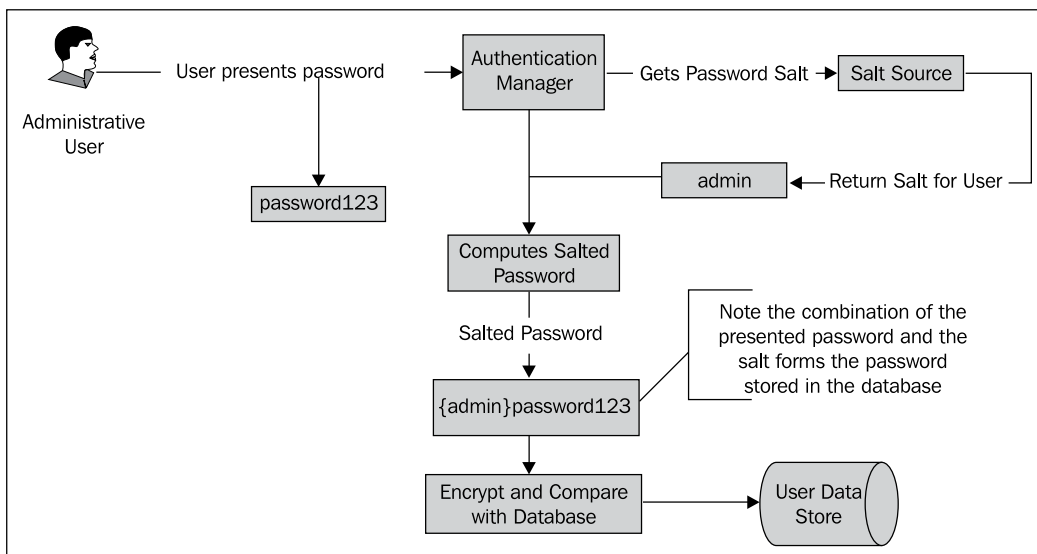
> Remember that because the salt is added to the plaintext password, the salt can't be one-way encrypted—the application needs to be able to look up or derive the appropriate salt value for a given user's record in order to authenticate the user!

Spring Security provides us with an interface, `o.s.s.authentication.dao.SaltSource`, which defines a method to return a salt value from the `UserDetails` object, along with two out of the box implementations:

- `SystemWideSaltSource` defines a single, static salt used for every password. This is not significantly more secure than an unsalted password.
- `ReflectionSaltSource` uses a bean property of the `UserDetails` object to get the salt value for that user's password.

Since salt values should be derivable from, or stored with each user record, `ReflectionSaltSource` is the out of the box implementation that most implementers typically use.

Configuring a salted password

As with configuring basic password encryption in the previous exercise, adding elements to support a salted password requires changes in both the bootstrap code and the `DaoAuthenticationProvider` We can examine how the workflow of salted passwords changes bootstrap and authentication by revising the following diagram occurred earlier in the book:
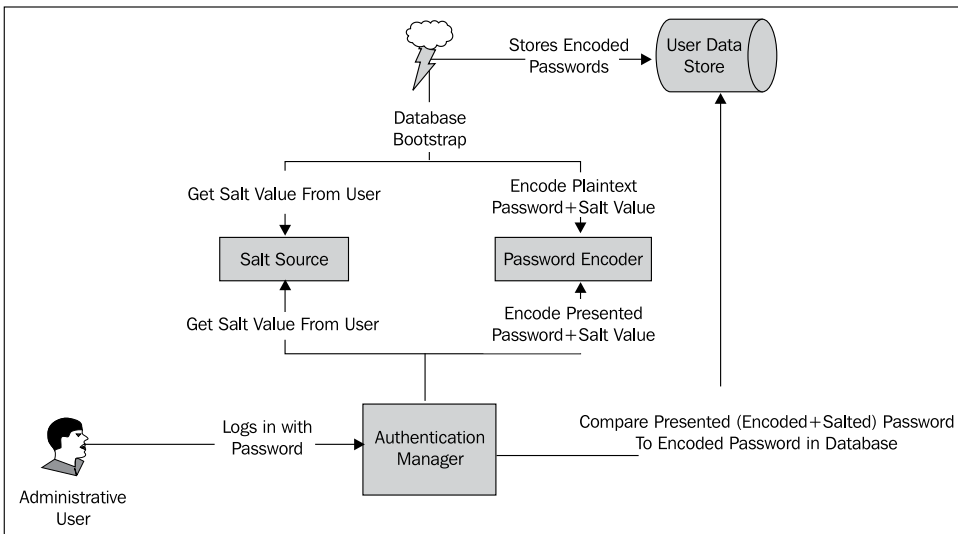
Let's add a level of password security by configuring the `ReflectionSaltSource` to salt our passwords!

Declaring the SaltSource Spring bean

In `dogstore-base.xml`, add a bean declaration for the `SaltSource` implementation we're using:

```
<bean class="org.springframework.security.authentication.
           dao.ReflectionSaltSource" id="saltSource">
  <property name="userPropertyToUse" value="username"/>
</bean>
```

We're configuring the salt source to use the `username` property, but this is a short-term implementation that we'll correct in a later exercise. Can you think why this might not be a good salt value?

Wiring the PasswordEncoder to the SaltSource

We'll need to hook up the `SaltSource` to the `PasswordEncoder`, so that the credentials the user presents upon login can be appropriately salted, prior to comparison with stored values. This is done by adding a new declaration in `dogstore-security.xml`:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider user-service-ref="jdbcUserService">
    <password-encoder ref="passwordEncoder">
      <salt-source ref="saltSource"/>
    </password-encoder>
  </authentication-provider>
</authentication-manager>
```

You'll note that if you start up the application at this point, you won't be able to log in. Just as in the previous exercise, the bootstrap database password encoder bean needs to be modified to include the `SaltSource`.

Augmenting DatabasePasswordSecurerBean

We'll add a bean reference to the `DatabasePasswordSecurerBean`, as well as a reference to the `UserDetailsService` so that we can get the appropriate password salt for the user:

```
public class DatabasePasswordSecurerBean extends JdbcDaoSupport {
  @Autowired
  private PasswordEncoder passwordEncoder;
```

```
    @Autowired
    private SaltSource saltSource;
    @Autowired
    private UserDetailsService userDetailsService;

    public void secureDatabase() {
      getJdbcTemplate().query("select username, password from users",
                              new RowCallbackHandler(){
        @Override
        public void processRow(ResultSet rs) throws SQLException {
          String username = rs.getString(1);
          String password = rs.getString(2);
          UserDetails user =
            userDetailsService.loadUserByUsername(username);
          String encodedPassword =
            passwordEncoder.encodePassword(password,
            saltSource.getSalt(user));
          getJdbcTemplate().update("update users set password = ?
            where username = ?",
          encodedPassword,
            username);
          logger.debug("Updating password for username:
                      "+username+" to: "+encodedPassword);
        }
      });
    }
  }
```

Recall that the `SaltSource` relies on a `UserDetails` object to generate the salt value. At this point, as we don't have the database row mapped to a `UserDetails` object, we'll have to make a request to the `UserDetailsService` (our `CustomJdbcDaoImpl`) to look up the `UserDetails` based on the username from the SQL query response.

At this point, we should be able to start the application and properly log into the system. If you try adding a new user with the same password (for example, `admin`) to the bootstrap SQL script, you'll note that the password generated for the user is different, because we're salting the passwords with the username. This makes the passwords much more secure in the unlikely event that a malicious user is able to read the passwords from the database. However, you may have some ideas as to why using the username isn't the most secure possible salt—we'll explore this in a later exercise.

Enhancing the change password functionality

One important change that we'll need to make is to update the `changePassword` functionality to refer to the password encoder implementation as well. This is as simple as adding appropriate bean references to the `CustomJdbcDaoImpl` class, and minor code changes in the `changePassword` method:

```
public class CustomJdbcDaoImpl extends JdbcDaoImpl {
  @Autowired
  private PasswordEncoder passwordEncoder;
  @Autowired
  private SaltSource saltSource;

  public void changePassword(String username, String password) {
    UserDetails user = loadUserByUsername(username);
    String encodedPassword = passwordEncoder.encodePassword
      (password, saltSource.getSalt(user));
    getJdbcTemplate().update(
      "UPDATE USERS SET PASSWORD = ? WHERE USERNAME = ?",
      encodedPassword, username);
  }
```

The use of the `PasswordEncoder` and `SaltSource` here will ensure that the user's password is properly salted when changed. Curiously, use of a `PasswordEncoder` and `SaltSource` is not supported by the `JdbcUserDetailsManager`, so if you are using `JdbcUserDetailsManager` as a baseline for customization, you will need to override quite a bit of code.

Configuring a custom salt source

As mentioned when we first configured password salting, `username` is a feasible, but not extremely desirable, choice for a password salt. The reason for this is that username is a salt that is under direct control of the user. If users are provided with the ability to change their username, it would be possible for malicious users to continuously change their user names—thus re-salting their passwords—and possibly determine how to construct a falsely encrypted password.

More secure still would be to have a property of the `UserDetails` which is chosen by the system, and never visible to, or modifiable by the user. We'll add a property to the `UserDetails` object that is populated at random when a user is created. This property will become the user's salt.

Extending the database schema

We'll need to ensure that the salt is stored in the database along with the user record, so we'll add a column to the default Spring Security database schema, defined in `security-schema.sql`:

```
create table users(
   username varchar_ignorecase(50) not null primary key,
   password varchar_ignorecase(50) not null,
   enabled boolean not null,
   salt varchar_ignorecase(25) not null
   );
```

Next, add the bootstrap salt values to the `test-users-groups-data.sql` script:

```
insert into users(username, password, enabled, salt) values ('admin','
admin',true,CAST(RAND()*1000000000 AS varchar));
insert into users(username, password, enabled, salt) values ('guest','
guest',true,CAST(RAND()*1000000000 AS varchar));
```

Remember, we're replacing the `insert` statements that are already there with these new ones. Note that we've chosen salt values based on random number generation—any pseudo-random salt selection of your choice can be just as effective.

Tweaking configuration of the CustomJdbcDaoImpl UserDetails service

Following similar steps as the custom schema exercise earlier in this chapter, we'll add a configuration change to the query used to retrieve users from the database in order to pick up the additional 'salt' column. We'll modify the `CustomJdbcDaoImpl` bean definition in `dogstore-security.xml`:

```
<beans:bean id="jdbcUserService"
   class="com.packtpub.springsecurity.security.CustomJdbcDaoImpl">
   <beans:property name="dataSource" ref="dataSource"/>
   <beans:property name="enableGroups" value="true"/>
   <beans:property name="enableAuthorities" value="false"/>
   <beans:property name="usersByUsernameQuery">
     <beans:value>select username,password,enabled,
                 salt from users where username = ?
     </beans:value>
   </beans:property>
</beans:bean>
```

Overriding the baseline UserDetails implementation

We'll need a `UserDetails` implementation that tracks the salt value stored with the user record in the database. Simply overriding the Spring standard `User` class is sufficient for our purposes. Remember to add a getter and setter for the salt, so that the `ReflectionSaltSource` password salter can find the right property.

```
package com.packtpub.springsecurity.security;
// imports
public class SaltedUser extends User {
  private String salt;
  public SaltedUser(String username, String password,
                    boolean enabled,
    boolean accountNonExpired, boolean credentialsNonExpired,
    boolean accountNonLocked, List<GrantedAuthority>
            authorities, String salt) {
    super(username, password, enabled,
          accountNonExpired, credentialsNonExpired,
    accountNonLocked, authorities);
    this.salt = salt;
  }
  public String getSalt() {
    return salt;
  }
  public void setSalt(String salt) {
    this.salt = salt;
  }
}
```

Even though we are extending the `UserDetails` to capture a salt field, the process would be the same if we wanted to store additional information about the user from a backing store. Extension of the `UserDetails` object is commonly done in conjunction with the implementation of a custom `AuthenticationProvider`. We'll see an example of this in *Chapter 6, Advanced Configuration and Extension*.

Extending the functionality of CustomJdbcDaoImpl

We need to override the methods of `JdbcDaoImpl`, responsible for instantiating the `UserDetails` implementation that sets the default values for `User`. This occurs while loading the `User` from the database, and then copying the `User` into the instance that actually gets returned from the `UserDetailsService`:

```
public class CustomJdbcDaoImpl extends JdbcDaoImpl {
  public void changePassword(String username, String password) {
    getJdbcTemplate().update(
```

```
                "UPDATE USERS SET PASSWORD = ? WHERE USERNAME = ?"
                 password, username);
        }

        @Override
        protected UserDetails createUserDetails(String username,
                    UserDetails userFromUserQuery,
                    List<GrantedAuthority> combinedAuthorities) {
                    String returnUsername = userFromUserQuery.getUsername();

                    if (!isUsernameBasedPrimaryKey()) {
                        returnUsername = username;
                }

          return new SaltedUser(returnUsername,
            userFromUserQuery.getPassword(),userFromUserQuery.isEnabled(),
            true, true, true, combinedAuthorities,
            ((SaltedUser) userFromUserQuery).getSalt());
        }

        @Override
        protected List<UserDetails> loadUsersByUsername(String username) {
                return getJdbcTemplate().
                query(getUsersByUsernameQuery(),
                new String[] {username},
                new RowMapper<UserDetails>() {
                  public UserDetails mapRow(ResultSet rs, int rowNum)
                                    throws SQLException {
                      String username = rs.getString(1);
                      String password = rs.getString(2);
                      boolean enabled = rs.getBoolean(3);
                      String salt = rs.getString(4);
                      return new SaltedUser(username, password,
                        enabled, true, true, true,
                        AuthorityUtils.NO_AUTHORITIES, salt);
                  }
                });
        }
    }
```

The methods `createUserDetails` and `loadUsersByUsername` are adapted from the superclass methods—the changes from the superclass methods are highlighted in the code listing. With these changes, you should be able to restart the application and have extra secure, randomly salted passwords. You may wish to add logging, and experiment, to see how the encrypted values change on every run of the application, as the bootstrap users are loaded.

Keep in mind that although this example illustrated the addition of a simple field to the `UserDetails` implementation, this approach can be used as a baseline to a highly customized `UserDetails` object which fits the business needs of your application. For the purposes of JBCP Pets, the auditors are now sufficiently happy with the security of passwords in the database—a job well done!

Moving remember me to the database

Something that you may have noticed by now, with our remember me implementation, is that it works very well until the application server is restarted, at which point the user's session is forgotten. This could be inconvenient for our users, who shouldn't have to pay attention to the maintenance and ups and downs of JBCP Pets.

Fortunately, Spring Security provides the capability to persist `rememberme` tokens any store implementing the `o.s.s.web.authentication.rememberme.PersistentTokenRepository` interface, and ships with a JDBC implementation of this interface.

Configuring database-resident remember me tokens

Modifying our remember me configuration at this point to persist to the database is surprisingly trivial. The Spring Security configuration parser will recognize a new `data-source-ref` attribute on the `<remember-me>` declaration and simply switch implementation classes for `RememberMeServices`. Let's review the steps required to accomplish this now.

Adding SQL to create the remember me schema

We'll place the SQL file containing the expected schema definition on the classpath in `WEB-INF/classes`, alongside the other bootstrap SQL scripts we've been working with. Let's call this SQL script `remember-me-schema.sql`:

```
create table persistent_logins (
  username varchar_ignorecase(50) not null,
  series varchar(64) primary key,
  token varchar(64) not null,
  last_used timestamp not null);
```

Adding new SQL script to the embedded database declaration

Next, add a reference to the new SQL script that we created in the `<embedded-database>` declaration in `dogstore-security.xml`:

```
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:security-schema.sql"/>
  <jdbc:script location="classpath:remember-me-schema.sql"/>
  <jdbc:script location="classpath:test-users-groups-data.sql"/>
</jdbc:embedded-database>
```

Configuring remember me services to persist to the database

Finally, we'll need to make some brief configuration changes to the `<remember-me>` declaration to point it to the data source we're using:

```
<http auto-config="true" use-expressions="true"
      access-decision-manager-ref="affirmativeBased">
  <intercept-url pattern="/login.do" access="permitAll"/>
  <intercept-url pattern="/account/*.do"
          access="hasRole('ROLE_USER') and fullyAuthenticated"/>
  <intercept-url pattern="/*" access="hasRole('ROLE_USER')"/>
  <form-login login-page="/login.do" />
  <remember-me key="jbcpPetStore" token-validity-seconds="3600"
          data-source-ref="dataSource"/>
  <logout invalidate-session="true" logout-success-url=""
          logout-url="/logout"/>
</http>
```

That's all we need. Now, if we restart the application, it will no longer forget users who previously had a valid remember me cookie set.

Are database-backed persistent tokens more secure?

You may recall that the `TokenBasedRememberMeServices` we implemented in Chapter 3 use MD5 hashing across a series of user-related data to encode the cookie securely, in a way that would be hard (but not impossible) to tamper with. The `o.s.s.web.authentication.rememberme.PersistentTokenBasedRememberMeServices` class implements handling of persistent tokens and handles token security with a validation method, which handles potential tampering slightly differently.

`PersistentTokenBasedRememberMeServices` creates a unique **series identifier** per user, with unique tokens within that series, as the user continues to interact and be authenticated by the site. The combination of series and token are stored in the cookie, and used to authenticate the user against the persistent token store. Both series and token are randomly generated, and of configurable length, making the likelihood of successful brute force guessing by a malicious attacker extremely small.

Just like `TokenBasedRememberMeServices`, persistent tokens may be compromised by cookie theft or other man-in-the-middle techniques. The use of a custom subclass incorporating the IP address into the persistent token, as well as forcing username and password authentication for sensitive areas of the site, are still advised when using persistent tokens.

Securing your site with SSL

It is highly likely that you have both heard of and used SSL encryption in your daily online life. The **Secure Sockets Layer** (**SSL**) protocol, and its successor, **Transport Layer Security** (**TLS**), are used to provide transport level security for HTTP transactions over the web—these are known as **HTTP Secure** (**HTTPS**) transactions.

In summary, SSL and TLS are used to secure the raw HTTP data being transmitted between the browser client and the web server in a way that is transparent to the user. As a developer, however, it's of critical importance to plan for the use of SSL in the design of a secure website. Spring Security provides a number of configuration options that allow for flexible incorporation of SSL in your web application.
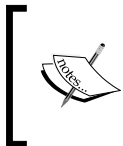
> Although SSL and TLS are different protocols (TLS being the more mature and recent iteration of the protocol), most people are familiar with the term SSL, and as such we will use this term to refer to both SSL and TLS protocols for the remainder of this book.

While a detailed examination of the mechanics of the SSL protocol are beyond the scope of this book, several excellent books and technical papers exist, which describe the specifications and protocols in very fine detail (you may want to start with *RFC 5246, The Transport Layer Security (TLS) Protocol Version 1.2*, at `http://tools.ietf.org/html/rfc5246`).

Setting up Apache Tomcat for SSL

First and foremost, if you're planning on following along with the SSL-related examples, you'll need to configure your application server to support SSL connections. For Apache Tomcat, this is relatively easy; if you are using another application server, please consult the relevant sections of the documentation.

Generating a server key store

We'll need to use the Java `keytool` command to generate a key store. Open a command prompt and enter the following command:

```
keytool -genkeypair -alias jbcpserver -keyalg RSA -validity 365
  -keystore tomcat.keystore -storetype JKS
```

Follow the prompts using the following directions. Enter the password **password** for the key store and private key password.

```
What is your first and last name?
  [Unknown]:  JBCP Pets Admin
What is the name of your organizational unit?
  [Unknown]:  JBCP Pets
What is the name of your organization?
  [Unknown]:  JBCP Pets
What is the name of your City or Locality?
  [Unknown]:  Anywhere
What is the name of your State or Province?
  [Unknown]:  NH
What is the two-letter country code for this unit?
  [Unknown]:  US
Is CN=JBCP Pets Admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US
correct?
  [no]:  yes
```

This will result in the creation of a file named `tomcat.keystore` in the current directory. This is the key store that will be used to enable Tomcat SSL.

> Be aware that the command `genkeypair` was called (`genkey` in pre-Java 6 releases of `keytool`).

Remember the location of this file for the next step.

Configuring Tomcat's SSL Connector

In Apache Tomcat's `conf` directory, open `server.xml` in an XML editor (Eclipse or equivalent will be fine), and uncomment or add the SSL `Connector` declaration. It should look as follows:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
  maxThreads="150" scheme="https" secure="true"
```

```
        sslProtocol="TLS"
        keystoreFile="conf/tomcat.keystore"
        keystorePass="password"/>
```

Ensure the `tomcat.keystore` file you created in the previous step is copied to the `conf` directory of the Tomcat installation. After this configuration, the Tomcat server can be started, and the JBCP Pets application should be accessible on a secure port, at `https://localhost:8443/JBCPPets/`.

Take care to include the `https` and not `http` — depending on the browser, this problem may go undetected, and you may find yourself scratching your head as to why you are not seeing the JBCP Pets home page.

Automatically securing portions of the site

It seems reasonable to assume that if you've gone to the trouble of setting up SSL to secure your customers' data, you would always want certain areas of the site to fall under the umbrella of SSL protection. Fortunately, Spring Security makes this easy, with the simple step of adding a configuration property to the `<intercept-url>` declarations.

The attribute `requires-channel` can be added to any `<intercept-url>` declaration to require that any URL matching the pattern is required to pass over a specific protocol (HTTP, HTTPS, or any). If we secure the JBCP Pets site in this way, the configuration would be as follows:

```
    <http auto-config="true" use-expressions="true">
      <intercept-url pattern="/login.do" access="permitAll"
                     requires-channel="https"/>
      <intercept-url pattern="/account/*.do"
                     access="hasRole('ROLE_USER') and fullyAuthenticated"
                     requires-channel="https"/>
      <intercept-url pattern="/*" access="permitAll"
                     requires-channel="any"/>
      <!-- ... -->
    </http>
```

If we start the application at this point, we see the following behavior:

- Access to the login page and the account page, now requires HTTPS, and the browser will automatically redirect the user from the non-secure (HTTP) URL to the secure URL. For example, attempts to access `http://localhost:8080/JBCPPets/login.do` will be redirected to `https://localhost:8443/JBCPPets/login.do`.

- Once the user has been switched to the HTTPS-secured URL, if he accesses a URL that isn't required to use HTTPS, he can still remain on HTTPS.

We can imagine that the important security benefits of this type of configuration—most modern application servers use the `secure` indicator on session cookies, so enforcing the login page to be secure (if this is where your application's sessions are first allocated) will ensure that the session cookie is transferred securely, and is less subjected to session hijacking. Additionally, this wires rules about SSL encryption directly into your security declarations, so it's very easy to ensure that all sensitive pages in the application are properly and thoroughly secured.

The functionality of automatically redirecting users to the appropriate protocol (HTTP or HTTPS) is implemented by the addition of another servlet filter towards the front of the Spring Security filter chain (just after the `SecurityContextPersistenceFilter`). The `o.s.s.web.access.channel.ChannelProcessingFilter` is automatically placed in the filter chain if any URL declarations are indicated to require a particular protocol, using the `requires-channel` attribute.

The interaction of the `ChannelProcessingFilter` in the processing of the request is illustrated in the following figure:
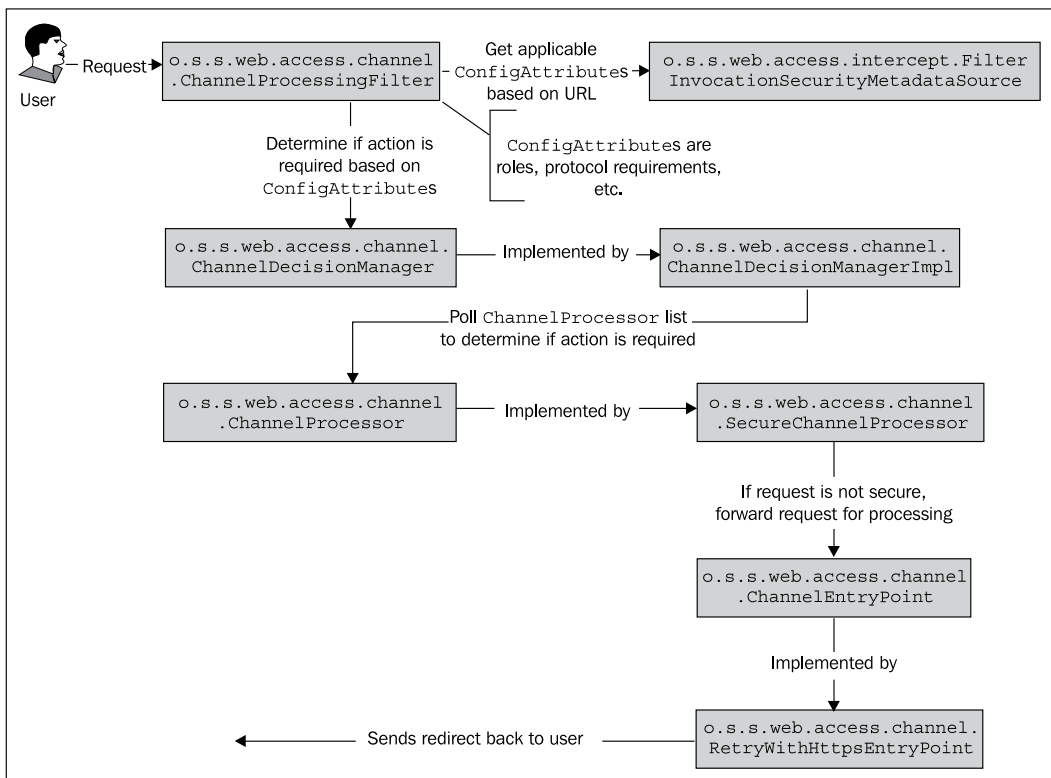
The design of the `ChannelProcessingFilter` lends itself to extension or augmentation, should your application require more complex logic than ships out of the box. Please note that although we illustrate only the `SecureChannelProcessor` and `RetryWithHttpsEntryPoint` implementation classes in the diagram, similar classes exist to validate and act on URLs that are declared to require HTTP.

It's important to note that, since the `ChannelEntryPoint` performs simple URL rewriting with an HTTP 302 redirect, it is of course not possible to use this technique to redirect POST URLs (although typically a POST should not transition between secure and insecure protocols anyway, since most browsers will warn against this behavior).

Secure port mapping

Certain environments may have HTTP or HTTPS ports other than the standard defaults of 80/443 or 8080/8443. In this case, you must augment your application's configuration to include explicit port mappings, so that the `ChannelEntryPoint` implementations can determine which port to use when redirecting users to secure or non-secure URLs.

This is trivial to do with the additional configuration element `<port-mappings>`, which allows for specification of additional HTTP or HTTPS pairs in addition to the defaults:

```
<port-mappings>
  <port-mapping http="9080" https="9443"/>
</port-mappings>
```

The use of port mappings can be especially important if your application server is behind a reverse proxy.

Summary

In this chapter, we have:

- Introduced the configuration to support a permanent security data store using a JDBC-compatible database.

- Configured JBCP Pets to use the database for user authentication and highly secure password storage, using password encryption and salting techniques.

- Evaluated techniques for managing users persisted to a database using JDBC.

- Configured user assignment into security groups, conferring roles, rather than direct role assignment. This increases the manageability of the site and its user community.

- Explored the use of Spring Security with a legacy (non-default) database schema.

- Examined configuration and application design techniques used to incorporate HTTPS to increase the security of data transferred to and from secured areas of the application.

In the next chapter, we'll explore some more advanced authorization capabilities of Spring Security, and introduce the Spring Security JSP tag library for fine-grained authorization.

5
Fine-Grained Access Control

Up to this point, we've updated the JBCP Pets site with more user-friendly functionality, including a custom login page, as well as change password, and remember me features.

In this chapter, we will first look at techniques for planning application security and user/group assignment. After this, we will examine two ways to implement **fine-grained authorization**—authorization that may affect portions of a page of the application. Next, we will look at Spring Security's approach for securing the business tier through method annotation and use of aspect-oriented programming. Finally, we will review an interesting capability of annotation-based security which allows for role-based filtering on collections of data.

During the course of this chapter, we'll learn:

- The basic techniques for planning web application security and group management, using a combination of off-the-shelf tools and critical thinking

- Configuring and experimenting with different methods of performing in-page authorization checks on content, given the security context of a user request

- Performing configuration and code annotation to make caller pre-authorization a key part of our application's business tier security

- Several alternative approaches to implementing method level security, and review the pros and cons of each type

- Implementing data-based filters on `Collections` and `Arrays` using method level annotations

As this chapter involves concepts that are beyond many of the isolated techniques we've seen thus far, we've made a number of source code changes on your behalf in order to increase the breadth of the website, and separate it into a true three-tier system. While these changes may be of interest to you, they don't directly relate to Spring Security, so we'll omit the details of the changes. Just don't be startled when you see many more files associated with this chapter's source code!

Re-thinking application functionality and security

At this point, we'd like to revisit the authorization model and flow of the JBCP Pets application. We feel that we've got a highly secure application, but the flow of the application is not particularly suitable for a public facing e-commerce site. This has much to do with the fact that access to any page in the application (except for the login page) requires that the user has a valid account and login—this is not conducive to users browsing and shopping!

Planning for application security

It's usually the domain of the product management and the security officer, in conjunction with engineering, to evaluate the desired functionality of the site against the planned user communities and their needs. This planning process—if performed effectively—relies on worksheets and diagramming to thoroughly cover the roles and groups that the application will comprise. We'll spend a bit of time walking through our planned expanded functionality for the JBCP Pets site to illustrate how this process could work. The thought that you put into the security planning process in any project you work on will pay dividends as you go into development—endeavor to build a security profile for each and every page and business service in your application design process!

Planning user roles

For JBCP Pets, we'll use the following table to map user classes to roles (Spring Security `GrantedAuthority` values). Some of these are new roles that could be used to classify different segments of the user population than we've used up to this point.

User class	Description	Roles
Guest	A user who isn't a remembered or authenticated user.	None (anonymous)
Consumer / Customer	A user who has established an account and may or may not have completed a purchase on the site.	`ROLE_CUSTOMER` `ROLE_USER`
Customer w/ Completed Purchase	A user who has completed at least one purchase on the site.	`ROLE_PURCHASER` `ROLE_USER`
Administrator	An administrative user responsible for user account management, and so on.	`ROLE_ADMIN` `ROLE_USER`
Supplier	A product supplier allows access to manage inventory of their products.	`ROLE_SUPPLIER` `ROLE_USER`

Using these declared user classes and roles, we can now match these roles in a coarse-grained way up against the functional definition of the site. There are many ways that this can be done—here are a few ideas that we've found helpful in the past:

- Use Microsoft Visio and Venn diagrams to indicate the overlap between functionality and user groups (remember that we used this to a very limited capacity in *Chapter 2*, *Getting Started with Spring Security*). This technique is visually effective for very small application domains or coarse-grained analysis.

- Diagram individual pages, and annotate them with the user classes or roles which are allowed to access each page. While not as visually accessible, this approach tends to be very accurate. We illustrate an example of this in the next section.

- Use sticky note modeling and a whiteboard or sketch board. In this type of exercise, product planners sketch out areas of a whiteboard to represent different user roles, and affix sticky notes representing product functional areas to each section of the whiteboard.

It can often be easier to do initial security planning in a non-digital format, as it's quite common for group discussion to lead to significant changes in the application security profile that are easier to adjust in a non-digital medium. Typically, security planning at this level doesn't get down to the level of detail of individual pages or portions of pages, but rather functional "chunks" of the application.

Planning page-level security

The next level of detail in security planning is security around page-level elements. First, planning of site-wide page features is essential, to ensure consistency in users' experience in available features from page to page as they traverse the site. Most sites will already have a site-wide templating facility in place, as simple as `jsp:include` directives (as are used in JBCP Pets), or more sophisticated, such as Apache Tiles 2.

Page-level security planning often dovetails with the process that you're already using for user experience planning for the site—many companies use Microsoft Visio or Adobe Dreamweaver for low-fidelity site planning, or more complex tools such as Axure RP. Whatever the tool, make sure that planning of feature availability as it relates to security is incorporated in the earliest designs of the site. Your UI architect and graphic designer will appreciate discussion of elements which may or may not appear, depending on the user's roles. Understanding the optionality of page elements will allow good UI planners to design the page with an appropriate flexible layout model to ensure that the page looks good regardless of the user's authorization.

> **Using the Right tools for the Job**
>
> A UI architect friend of mine turned me on to the excellent "Sketchy" shape set for Visio, available at: `http://www.guuui.com/ issues/02_07.php`. For those familiar with Visio, this is a visually appealing way to develop accurate, but low-fidelity mockups using a tool that many users already understand Although no open source Visio-compatible software currently exists, similar applications such as Dia (`http://projects.gnome.org/dia/`) or OpenOffice Draw (`http://www.openoffice.org/product/draw.html`) are available for most platforms.

A sample Visio diagram annotated with security information might appear as follows:
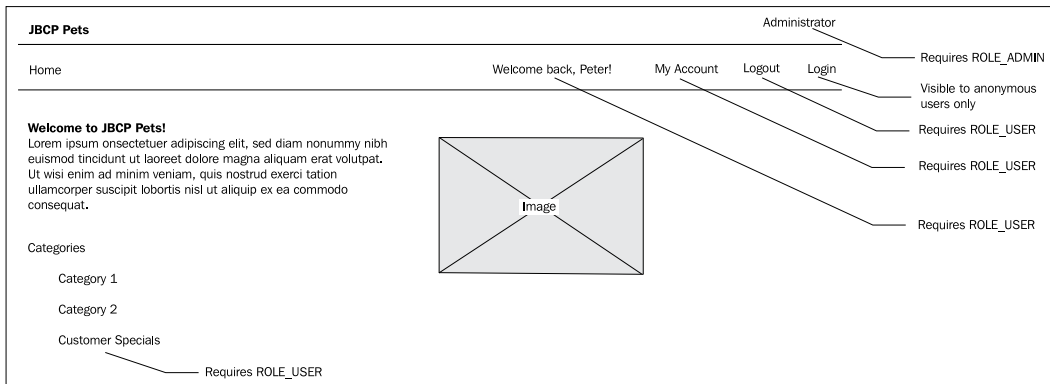
You can see that we don't require too much detail to capture the relevant security information, but the intention of the diagram is instantly accessible to anyone (even non-technical users) who might review it.

Methods of Fine-Grained authorization

Fine-grained authorization refers to the availability of application features based on the context of a particular user's request. Unlike coarse-grained authorization we explored in *Chapters 2*, *Getting Started with Spring Security*, *Chapter 3*, *Enhancing the User Experience* and *Chapter 4*, *Securing Credential Storage*, fine-grained authentication typically refers to selective availability of portions of a page, rather than restricting access to a page entirely. Most real-world applications will spend a considerable amount of time in the details of fine-grained authorization planning.

Spring Security provides us with two methods of selective display of functionality:

- **Spring Security JSP Tag Libraries** allow conditional access declarations to be placed within a page declaration itself, using standard JSP tag library syntax.

- Checking user authorization in an MVC application's **controller** layer allows the controller to make an access decision and bind the results of the decision to the **model** data provided to the **view**. This approach relies on standard JSTL conditional page rendering and data binding and is slightly more complicated than the Spring Security tag libraries; however, it is more in line with standard web application MVC logical design.

Either of these approaches is perfectly valid when developing fine-grained authorization models for a web application. Let's explore how each approach is implemented through a JBCP Pets use case.

We'd like to use the results of our security planning to ensure the **Log Out** and **My Orders** links in the site-wide menu bar appear only for users who are logged in or have purchases (ROLE_USER and ROLE_CUSTOMER, respectively). We'd also like to ensure that the **Log In** link appears for users who are browsing the site as unauthenticated guests (users without the role ROLE_USER). We'll work through both approaches to add this functionality, starting with the Spring Security JSP Tag Library.

Using Spring Security Tag Library to conditionally render content

We saw in Chapter 3 that the Spring Security tag library can be used to access data contained in the `Authentication` object, and here we'll examine some other powerful functionality of the tag library. The most common functionality used in the Spring Security tag library is to conditionally render portions of the page, based on authorization rules. This is done with the `<authorize>` tag that functions similarly to the `<if>` tag in the `core` **JSTL** library, in that the tag's body will render depending on the conditions provided in the tag attributes. Let's use the `<authorize>` tag to conditionally render bits of the page.

Conditional rendering based on URL access rules

The Spring Security tag library provides functionality to render content based on the existing URL authorization rules that are already defined in the application security configuration file. This is done by the use of the `<authorize>` tag, with the `url` attribute.

For example, we could ensure that the **My Account** link is displayed only when appropriate, that is, for users who are actually logged into the site—recall that the access rules we've previously defined are as follows:

```
<intercept-url pattern="/account/*.do"
access="hasRole('ROLE_USER') and fullyAuthenticated"/>
```

So, the JSP code to conditionally display the **My Account** link would be as follows:

```
<sec:authorize url="/account/home.do">
  <c:url value="/account/home.do" var="accountUrl"/>
  <li><a href="${accountUrl}">My Account</a></li>
</sec:authorize>
```

This will ensure that the content of the tag is not displayed unless the user has sufficient privileges to access the stated URL. It is possible to further qualify the authorization check by HTTP method, by including the `method` attribute:

```
<sec:authorize url="/account/home.do"
    method="GET">
  <c:url value="/account/home.do" var="accountUrl"/>
  <li><a href="${accountUrl}">My Account</a> (with 'url' attr)</li>
</sec:authorize>
```

Using the `url` attribute to define authorization checks on blocks of code is convenient, because it abstracts knowledge of the actual authorization checks from your JSPs and keeps them in your security configuration file.

> Be aware that the HTTP method should match the case specified in your security `<intercept-url>` declarations otherwise they may not match as you expect. Also, note that the URL should always be relative to the web application context root (as your URL access rules are).

For many purposes, the use of the `<authorize>` tag's `url` attribute will suffice to correctly display link-or action-related content only when the user is allowed to see it. Remember that the tag need not only surround a link, but it could even surround a whole form, if the user doesn't have permission to submit it.

Conditional rendering based on Spring EL Expressions

An additional, more flexible method of controlling display of JSP content is available when the `<authorize>` tag is used in conjunction with a Spring Expression Language (SpEL) expression.

Recall from our initial exploration in Chapter 2 that SpEL provides a powerful expression language, which is further enhanced by Spring Security with constructs allowing expressions to be built around the current secured request. If we refine the previous example using SpEL expressions in the `<authorize>` tag to lock down access to the **My Account** link, it might be as follows:

```
<sec:authorize access="hasRole('ROLE_USER') and fullyAuthenticated">
    <c:url value="/account/home.do" var="accountUrl"/>
    <li><a href="${accountUrl}">My Account</a> (with 'access' attr)</li>
</sec:authorize>
```

The SpEL evaluation is performed by the same code behind the scenes as the expressions utilized in `<intercept-url>` access declaration rules (assuming expressions have been configured). Hence the same set of built-in functions and properties are accessible from expressions built using the `<authorize>` tag.

Either of these methods of utilizing the `<authorize>` tag provide powerful, fine-grained control over display of page contents based on security authorization rules.

Conditionally rendering the Spring Security 2 way

The two methods of utilizing the Spring Security JSP tag library listed previously are actually new in Spring Security 3, and are the preferred method of securing content within a page based on authorization rules; however, the same `<authorize>` tag supports several other methods of operation, which you may encounter in legacy code, or may in some cases better suit your needs.

Conditional display based on absence of a role

The **Log In** link should be displayed for users who are anonymous, that is, do not have the `ROLE_USER` role. The `<authorize>` tag supports this type of rule with the `ifNotGranted` attribute:

```
<sec:authorize ifNotGranted="ROLE_USER">
  <c:url value="/login.do" var="loginUrl"/>
  <li><a href="${loginUrl}">Log In</a></li>
</sec:authorize>
```

You'll note that if you now attempt to access the site as an anonymous user, you'll see a link pointing to the login form.

Conditional display based on any one of a list of roles

Similar to the last step, the **Log Out** link should be displayed for users who have an account with the site and are logged in. The `ifAnyGranted` attribute requires the user to have any one of several specified roles before rendering the tag body. We'll demonstrate the use of this form with the **Log Out** link:

```
<sec:authorize ifAnyGranted="ROLE_USER">
  <c:url value="/logout" var="logoutUrl"/>
  <li><a href="${logoutUrl}">Log Out</a></li>
</sec:authorize>
```

Note that the `ifAnyGranted` attribute allows a comma-separated set of roles to be specified to determine a proper match, the user needs to have only one of the roles for the tag body to be rendered.

Conditional display Based on all of a list of roles

Finally, the use of the `ifAllGranted` attribute requires that the user has every one of the roles listed in the tag attribute.

```
<sec:authorize ifAllGranted="ROLE_USER,ROLE_CUSTOMER">
  <c:url value="/account/orders.do" var="ordersUrl"/>
  <li><a href="${ordersUrl}">My Orders</a></li>
</sec:authorize>
```

We can see that there are many ways that the `authorize` tag can apply to different circumstances. Note that the three attributes we've exercised can be combined. For example the `ifNotGranted` and `ifAnyGranted` attributes can be combined to provide simple complex Boolean equations.

Using JSP Expressions

Each of the previous three methods of in-page authorization (`ifNotGranted`, `ifAnyGranted`, `ifAllGranted`) supports a JSP EL expression which will be evaluated to return an authorization `GrantedAuthority` (role, and so on.). This allows flexibility if the list of authorization requirements may change depending on earlier computation in the page.

Using controller logic to conditionally render content

Let's now adapt the examples we just implemented using the `<authorize>` tag to Java-based code. For the purposes of brevity, we'll implement only one of the examples, but it should be straightforward to follow along and see how to provide each of the examples when using controller-based authorization checks.

Adding conditional display of the Log In link

Instead of using the Spring Security `<authorize>` tag, we'll make the assumption that there's a Boolean variable in the model data available to the view that dictates whether or not it should show the **Log In** link. In traditional MVC fashion, the view needn't know why the model has this value, it just needs to act on it. We will use the **Java Standard Tag Library** (JSTL) `if` tag, along with a JSP EL expression, to conditionally display a portion of the page.

```
<c:if test="${showLoginLink}">
  <c:url value="/login.do" var="loginUrl"/>
  <li><a href="${loginUrl}">Log In</a></li>
</c:if>
```

Now let's review how we populate the model data from the controller with the following code.

Populating model data based on user credentials

The object model of our controllers is set up in a sensible object-oriented pattern, with all Spring MVC controllers extending from a single base class, `com.packtpub.springsecurity.web.controller.BaseController`. This allows us to place commonly used code in the `BaseController` and make it accessible to all controllers in the application. First, we'll put in a method that will get the `Authentication` implementation from the current request:

```
protected Authentication getAuthentication() {
  return SecurityContextHolder.getContext().getAuthentication();
}
```

Next, we'll add a method that will populate the `showLoginLink` model data value, using a Spring MVC-annotated method:

```
@ModelAttribute("showLoginLink")
public boolean getShowLoginLink() {
  for (GrantedAuthority authority : getAuthentication().
getAuthorities()) {
    if(authority.getAuthority().equals("ROLE_USER")) {
      return false;
    }
  }
  return true;
}
```

As this method is annotated with `@ModelAttribute`, it will automatically be executed by Spring MVC when any controller extending `BaseController` is invoked. We can simply repeat this design pattern for the remaining show/hide model data directives that we covered through the additional uses of the `authorize` tag.

What is the best way to configure in-page authorization?

The major advances in the Spring Security `<authorize>` tag in Spring Security 3 removed many of the concerns about the use of this tag in prior versions of the library. In many cases, the use of the `url` attribute of the tag can appropriately isolate the JSP code from changes in authorization rules. You should use the `url` attribute of the tag when:

- The tag is preventing display of functionality that can be clearly identified by a single URL
- The contents of the tag can be unambiguously isolated to a single URL

Unfortunately, in a typical application, the likelihood that you will be able to use the `url` version of the tag frequently is somewhat low. The reality is that applications are usually much more complex than this, and require more involved logic when deciding to render portions of a page.

Although it's tempting to use the Spring Security tag library to declare bits of rendered pages off-limits based on security criteria in the other ways we saw in this chapter, including the `if...Granted` and `access` (SpEL) methods, there are a number of reasons why (in many cases) this isn't a great idea:

- Complex conditions beyond role membership are not supported by the tag library. For example, if our application incorporated customized attributes on the `UserDetails` implementation, IP filters, geo-location, and so on—none of these would be supported using the standard `<authorize>` tag.

  These could, however, conceivably be supported by a custom JSP tag, or using SpEL expressions. Even in this case the JSP is more likely to be directly tied to business logic than is typically encouraged.

- The `<authorize>` tag must be referenced on every page it's used. This leads to potential inconsistencies between rule sets that are intended to be common, but may be spread across different physical pages. Good object oriented system design would suggest that conditional rule evaluations be located in only one place, and logically referred to from where they should be applied.

  It is possible (and we illustrate this using our common header JSP include) to encapsulate and reuse portions of JSP pages to reduce the occurrence of this type of problem, but it is virtually impossible to eliminate in a complex application.

- There is no way to validate the correctness of rules stated at compile time. Whereas compile-time constants can be used in typical Java-based object oriented systems, the JSP tag library requires (in typical use) hard-coded role names, where a simple typo might go undetected for some time.

  To be fair, such typos could be easily caught by comprehensive functional tests of the running application, but they are far easier to test using standard Java component unit testing techniques.

We can see that although the JSP-based approach for conditional content rendering is convenient, there are some significant downsides.

All of these issues can be solved through the use of code in controllers that can be used to push data into the application view model. Additionally, performing advanced authorization determinations in code allows the benefits of reuse, compile-time checks, and proper logical separation of model, view, and controller.

Securing the business tier

Our primary focus to this point in the book has been on securing the web-facing portion of the JBCP Pets application; however, in real-world planning of secured systems, equal attention should be paid to securing the service methods that allow users access to the most critical part of any system—its data.

Spring Security has the ability to add a layer of authorization (or authorization-based data pruning) to the invocation of any Spring-managed bean in your application. While many developers focus on web-tier security, business tier security is arguably just as important, as a malicious user may be able to penetrate the security of your web tier, or may be able to access services exposed through a non-UI front-end, such as a web service.

Let's examine the following logical diagram to see where we're interested in applying a secondary layer of security:
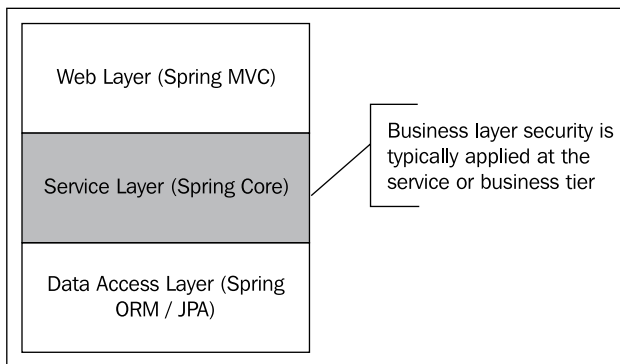
Spring Security has two main techniques for securing methods:

- **Pre-authorization** ensures that certain constraints are satisfied prior to execution of a method being allowed—for example, that a user has a particular `GrantedAuthority`, such as `ROLE_ADMIN`. Failure to satisfy the declared constraints means that the method call will fail.

- **Post-authorization** ensures that the calling principal still satisfies declared constraints after the method returns. This is used rarely, but can provide an extra layer of security around some complex interconnected business tier methods.

Pre-and post-authorization provide formalized support for what are generally termed **preconditions** and **postconditions** in classic object-oriented design. Preconditions and postconditions allow a developer to declare through runtime checks, that certain constraints around a method's execution must always hold true. In the case of security pre-and post-authorization, the business tier developer is making a conscious decision about the security profile of particular methods by encoding expected runtime conditions as part of an interface or class API declaration. As you may imagine, this can require a great deal of forethought to avoid unintended consequences!

The basics of securing business methods

Let's take some specific business methods in the JBCP Pets business tier and illustrate how to apply some typical rules to them.

As part of the reorganization of the JBCP Pets codebase into a three-tier design, we've abstracted the change password functionality seen in earlier chapters to the business tier. Instead of making a direct call from the web MVC controller to the JDBC DAO, we make the sensible choice of interposing a business service that can provide additional functionality should the need arise. This is illustrated in the following diagram:
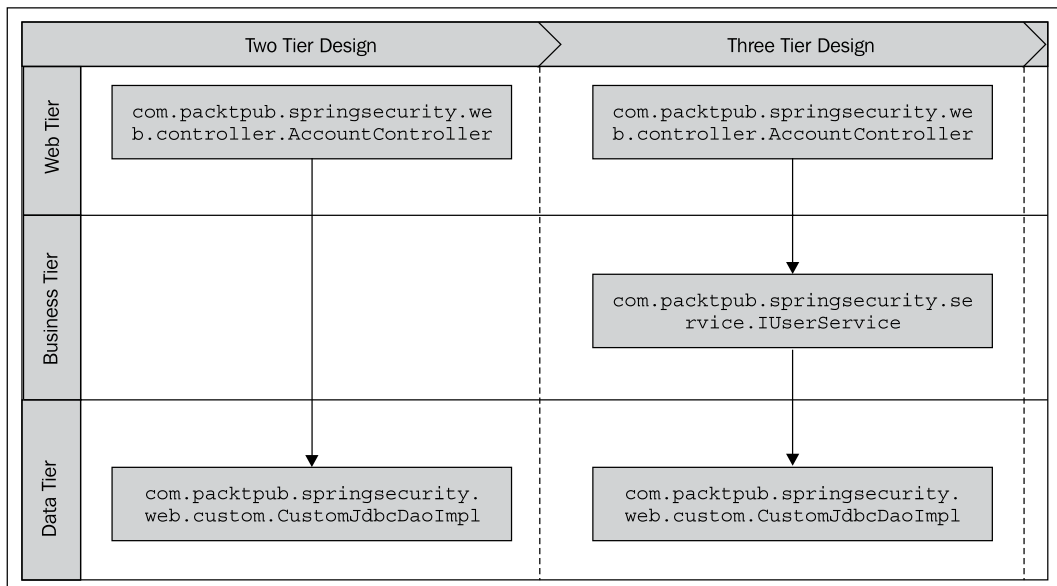
We can see in this example that the `com.packtpub.springsecurity.service.IUserService` interface would represent the business tier of the application architecture. This is an appropriate place for us to add method level security.

Adding @PreAuthorize method annotation

Our first design decision will be to augment method security at the business tier by ensuring that a user must be logged in as a valid user of the system before they are allowed to change a password. This is done with a simple annotation added to the method in the service interface definition as follows:

```
public interface IUserService {
  @PreAuthorize("hasRole('ROLE_USER')")
  public void changePassword(String username, String password);
}
```

This is all that is required to ensure that valid, authenticated users can access the change password function. Spring Security will use a runtime **aspect oriented programming** (**AOP**) **pointcut** to execute **before advice** on the method, and throw an `AccessDeniedException` if the security constraints aren't met.

Instructing Spring Security to use method annotations

We'll also need to make a one-time change to `dogstore-security.xml`, where we've got the rest of our Spring Security configuration. Simply add the following element right before the `<http>` declaration:

```
<global-method-security pre-post-annotations="enabled"/>
```

Validating method security

Don't believe it was that easy? Try changing the `ROLE_USER` declaration to `ROLE_ADMIN`. Now log in as user `guest` (password `guest`) and try to change your password. You'll see an error when you try to change your password:
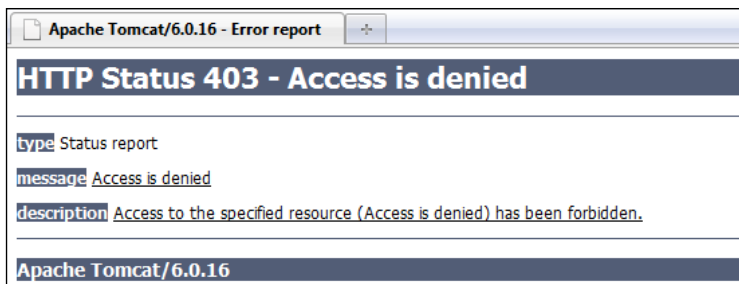
If you look at the Tomcat console, you'll see a very long stack trace, starting with:

```
DEBUG - Could not complete request
o.s.s.access.AccessDeniedException: Access is denied
at o.s.s.access.vote.AffirmativeBased.decide
at o.s.s.access.intercept.AbstractSecurityInterceptor.beforeInvocation
...
at $Proxy12.changePassword(Unknown Source)
at com.packtpub.springsecurity.web.controller.AccountController.
submitChangePasswordPage
```

Based on the access denied page, and the stack trace clearly pointing to the `changePassword` method invocation, we can see that the user was appropriately denied access to the business method because they lacked the `GrantedAuthority` of `ROLE_ADMIN`. You can test that the change password function is still allowed for the admin user.

Isn't it amazing that with a simple declaration in our interface, we're able to ensure that the method in question has been secured? Of course, we do not want the generic Tomcat `403` error page to be displayed in our production application—we'll update this in *Chapter 6*, *Advanced Configuration and Extension*, when we examine access denied handling.

Let's explore some other basic types of method security, and then we'll go behind the scenes to see how and why this works.

Several flavors of method security

In addition to the `@PreAuthorize` annotation, there are several other ways of declaring security pre-authorization requirements on methods. We can examine these different ways of securing methods, and then evaluate their pros and cons in different circumstances.

JSR-250 compliant standardized rules

JSR-250, *Common Annotations for the Java Platform*, defines a series of annotations, some security-related, which are intended to be portable across JSR-250 compliant runtime environments. The Spring Framework became compliant with JSR-250 as part of the Spring 2.x release, including the Spring Security Framework.

While the JSR-250 annotations are not as expressive as the Spring native annotations, they have the benefit that the declarations they provide are compatible across implementing Java EE application servers such as Glassfish, or service-oriented runtime frameworks such as Apache Tuscany. Depending on your application's needs and requirements for portability, you may decide that the trade-off of reduced specificity is worth the portability of the code.

To implement the rule we specified in the first example, we would make two changes, first in `dogstore-security.xml` file:

```
<global-method-security jsr250-annotations="enabled"/>
```

Secondly, the `@PreAuthorize` annotation would change to the `@RolesAllowed` annotation. As we might anticipate, the `@RolesAllowed` annotation does not support SpEL expressions, so it looks much like the URL authorization rules we covered early in Chapter 2. We edit the `IUserService` interface definition as such:

```
@RolesAllowed("ROLE_USER")
public void changePassword(String username, String password);
```

As with the previous exercise, if you don't believe this works, try changing `ROLE_USER` to `ROLE_ADMIN` and test it out!

Note that it's also possible to provide a list of allowed `GrantedAuthority` names, using standard Java 5 String array annotation syntax:

```
@RolesAllowed({"ROLE_USER","ROLE_ADMIN"})
public void changePassword(String username, String password);
```

There are also two additional annotations specified by JSR-250, `@PermitAll` and `@DenyAll` that function as you might expect, permitting and denying all requests to the method in question.

> **Annotations at the class level**
>
> Be aware that the method-level security annotations can also be applied at the class level as well! Method-level annotations, if supplied, will always override annotations specified at the class level. This can be helpful if your business needs dictate specification of security policies for an entire class at a time. Take care to use this functionality in conjunction with good comments and coding standards, so that developers are very clear about the security characteristics of a class and its methods.

We'll explore how the JSR-250 annotation style and Spring Security annotation style can co-exist in exercises later in this chapter.

Method security using Spring's @Secured annotation

Spring itself provides a simpler annotation style which is similar to the JSR-250 `@RolesAllowed` annotation. The `@Secured` annotation is functionally and syntactically the same as `@RolesAllowed`. The only notable difference is that processing of these annotations must be explicitly enabled with another attribute on the `<global-method-security>` element:

```
<global-method-security secured-annotations="enabled"/>
```

As `@Secured` functions the same as the JSR standard `@RolesAllowed` annotation, there's not really a compelling reason to use it in new code, but you may run across it in older Spring code.

Method security rules using Aspect Oriented Programming

The final technique for securing methods is quite possibly the most powerful method, with an additional benefit that it needn't require code modification at all. Instead, it uses aspect-oriented programming to declare a pointcut at a method or set of methods, with advice that performs checks for role membership when the pointcut matches. The AOP declarations are only present in the Spring Security XML configuration file, and do not involve any annotations.

The following is an example of a declaration protecting all service interface methods with administrative rights:

```
<global-method-security>
  <protect-pointcut access="ROLE_ADMIN" expression="execution(* com.
packtpub.springsecurity.service.I*Service.*(..))"/>
</global-method-security>
```

The pointcut expressions are supported under the hood with Spring AOP support via AspectJ. Unfortunately, Spring AspectJ AOP only supports a very small subset of the AspectJ pointcut expression language—refer to the Spring AOP documentation for more details on supported expressions and other important elements of programming with Spring AOP.

That said, be aware that it's possible to specify a series of pointcut declarations, targeting different roles and pointcut targets. Here's an example where we add a pointcut to target a method in our DAO:

```
<global-method-security>
  <protect-pointcut access="ROLE_USER" expression="execution(* com.
packtpub.springsecurity.dao.IProductDao.getCategories(..)) &amp;&amp;
args()"/>
  <protect-pointcut access="ROLE_ADMIN" expression="execution(* com.
packtpub.springsecurity.service.I*Service.*(..))"/>
</global-method-security>
```

Notice that the new pointcut we added uses some more advanced AspectJ syntax, illustrating Boolean logic and the other supported pointcut, `args`, which can be used to specify the type declaration of arguments.

Much as with other areas of Spring Security that allow a series of security declarations, AOP-style method security is processed top to bottom, so it's a good idea to write the pointcuts in most-specific to least-specific order.

Programming using AOP can be confusing for even seasoned developers. If you intend to use AOP heavily for security declarations, it is highly suggested that you review some of the very good books available on the subject, in addition to the Spring AOP reference documentation. AOP can be tricky to implement, especially diagnosing configuration errors if it doesn't work as you expect!

Comparing method authorization types

The following quick reference chart may assist you in selecting a type of method authorization checking to use:

Method Authorization Type	Specified As	JSR Standard	Allows SpEL Expressions
`@PreAuthorize` `@PostAuthorize`	Annotation	No	Yes
`@RolesAllowed` `@PermitAll` `@DenyAll`	Annotation	Yes	No
`@Secure`	Annotation	No	No
`protect-pointcut`	XML	No	No

Most Java 5 consumers of Spring Security will probably opt to use the JSR-250 annotations for maximum compatibility and reuse of their business classes (and relevant constraints) across the IT organization. Where needed, these basic declarations can be replaced with the annotations that tie the code to the Spring Security implementation itself.

If you are using Spring Security in an environment that doesn't support annotations (Java 1.4 or previous), unfortunately your choices are somewhat limited with method security enforcement. Even in this situation, the use of AOP provides a reasonably rich environment in which we can develop basic security declarations.

How does method security work?

The access decision mechanism for method security – whether or not a given request is allowed – is conceptually the same as the access decision logic for web request access. An `AccessDecisionManager` polls a set of `AccessDecisionVoters`, each of which can provide a decision to grant or deny access, or abstain from voting. The specific implementation of the `AccessDecisionManager` aggregates the voter decisions and arrives at an overall decision to allow the method invocation.

Web request access decision making is less complicated, due to the fact that the availability of `ServletFilters` makes interception (and summary rejection) of securable requests relatively straightforward. As method invocation can happen from anywhere, including areas of code not directly configured by Spring Security, the Spring Security designers chose to use a Spring-managed AOP approach to recognize, evaluate, and secure method invocations.

The following high level flow illustrates the main players involved in authorization decisions for method invocation:
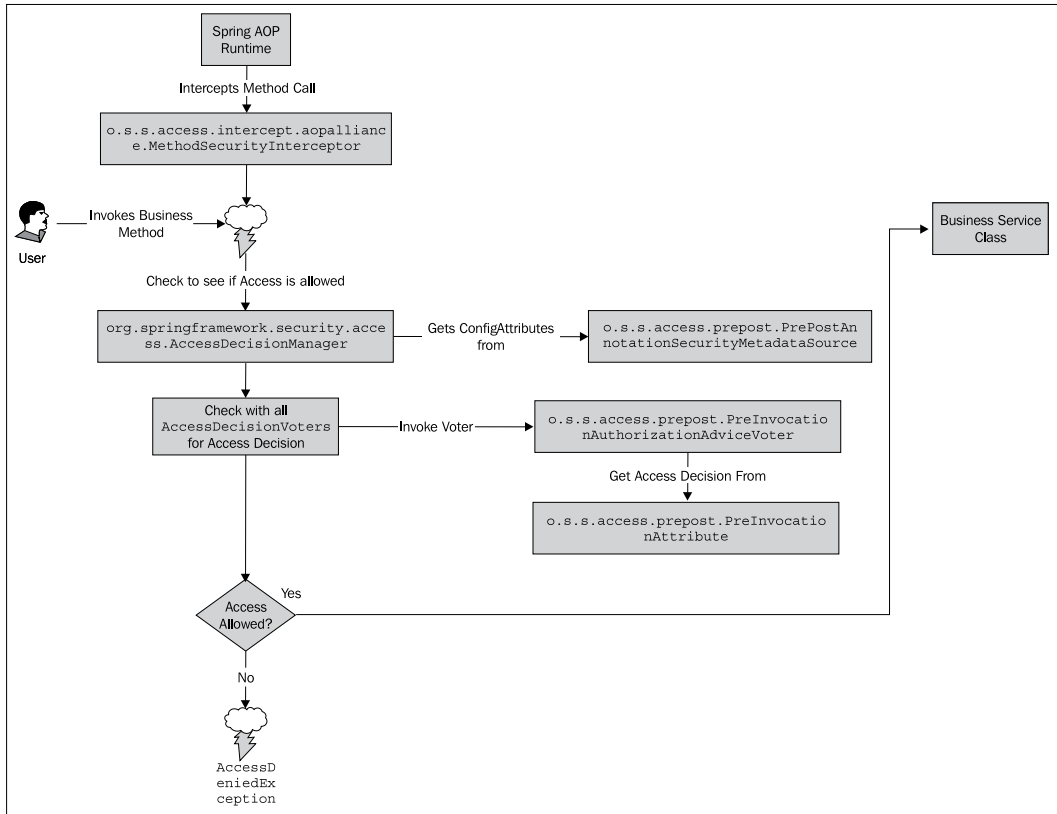
We can see that Spring Security's `o.s.s.access.intercept.aopalliance.` `MethodSecurityInterceptor` is invoked by the standard Spring AOP runtime to intercept method calls of interest. From here, the logic of whether or not to allow a method call is relatively straightforward, as per the previous flow diagram.

At this point, we might wonder about performance of the method security feature. Obviously, the `MethodSecurityInterceptor` couldn't be invoked for every method call in the application—so how do annotations on methods or classes result in AOP interception?

First of all, AOP proxying isn't invoked for all Spring-managed beans by default. Instead, if `<global-method-security>` is defined in the Spring Security configuration, a standard Spring AOP `o.s.beans.factory.config.BeanPostProcessor` will be registered which will introspect the AOP configuration to see if any AOP advisors indicate that proxying (and interception) is required. This workflow is standard Spring AOP handling (known as **AOP auto-proxying**), and doesn't inherently have functionality specific to Spring Security. All registered `BeanPostProcessors` run at initialization of the spring `ApplicationContext`, after all Spring Bean configurations have occurred.

The AOP auto-proxy functionality queries all registered `PointcutAdvisors` to see if there are AOP pointcuts that resolve to method invocations which should have AOP advice applied. Spring Security implements the `o.s.s.access.intercept.aopalliance.MethodSecurityMetadataSourceAdvisor` class, which examines any and all configured method security and sets up appropriate AOP interception. Take note that only interfaces or classes with declared method security rules will be proxied for AOP!

> Be aware that it is strongly encouraged to declare AOP rules (and other security annotations) on interfaces, and not on implementation classes. The use of classes, while available using CGLIB proxying with Spring, may unexpectedly change certain behavior of your application, and is generally less semantically correct than security declarations (through AOP) on interfaces.

`MethodSecurityMetadataSourceAdvisor` delegates the decision to affect methods with AOP advice to an `o.s.s.access.method.MethodSecurityMetadataSource` instance. The different forms of method security annotation each have their own `MethodSecurityMetadataSource` implementation, which is used to introspect each method and class in turn and add AOP advice to be executed at runtime.

The following diagram illustrates how this process occurs:
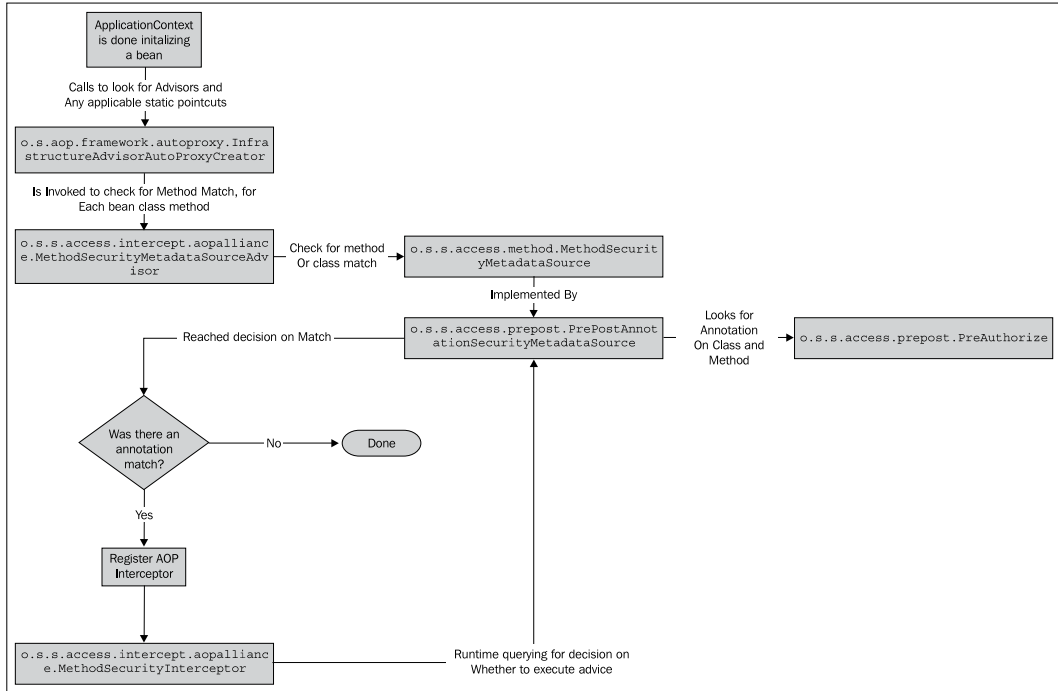
Depending on the number of Spring Beans configured in your application, and the number of secured method annotations you have, adding method security proxying may increase the time required to initialize your `ApplicationContext`. Once your Spring context is initialized, however, there is a negligible performance impact on individual proxied beans.

Advanced method security

Method security expressiveness doesn't stop at simple role checking. In fact, some method security annotations can utilize the full power of the Spring Expression Language (SpEL) that we first saw in Chapter 2 with the discussion about authorization rules in URL interceptors. This means that arbitrary expressions, with calculation, Boolean logic, and so on can be used.

Method security rules using bean decorators

An alternative form for declaring method security rules involves the use of declarative XML, which can be included within a Spring Bean definition. Although easier to read, this form of method security is far less expressive than pointcuts, and far less comprehensive than the annotation-based approaches that we've reviewed so far. Nonetheless, for certain types of projects, using an XML declarative approach may be sufficient for your needs.

We can experiment by replacing the rules we declared in the prior examples with XML-based declarations to secure the `changePassword` method. As we have used bean auto-wiring to this point, which is unfortunately not compatible with XML method decorators, we'll need to explicitly declare the service layer beans in order to demonstrate this technique.

The security decorators are part of the security XML namespace. We'll first need to include the security schema in our `dogstore-base.xml` file, which has been used to contain Spring Bean definitions to this point:

```
<?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:aop="http://www.springframework.org/schema/aop"
  xmlns:context="http://www.springframework.org/schema/context"
  xmlns:jdbc="http://www.springframework.org/schema/jdbc"
  xmlns:security="http://www.springframework.org/schema/security"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/aop http://www.
springframework.org/schema/aop/spring-aop-3.0.xsd
    http://www.springframework.org/schema/jdbc  http://www.
springframework.org/schema/jdbc/spring-jdbc-3.0.xsd
    http://www.springframework.org/schema/context http://www.
springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/security http://www.
springframework.org/schema/security/spring-security-3.0.xsd
  ">
```

Next (for the purposes of this exercise), remove any security annotations you may have on the `IUserService.changePassword` method.

Finally, declare the bean in Spring XML syntax, with the following additional decorator, which will declare that anyone wishing to invoke the `changePassword` method must be of `ROLE_USER`:

```
<bean id="userService" class="com.packtpub.springsecurity.service.
UserServiceImpl">
  <security:intercept-methods>
    <security:protect access="ROLE_USER" method="changePassword"/>
  </security:intercept-methods>
</bean>
```

As with the examples earlier in this chapter, this protection can be easily verified by changing `ROLE_USER` to `ROLE_ADMIN` and attempting to change your password using the guest user account.

Behind the scenes, the functionality of this type of method access protection uses a `MethodSecurityInterceptor` wired to a `MapBasedMethodSecurityMetadataSource`, which the interceptor uses to determine appropriate access `ConfigAttributes`. Unlike the more expressive SpEL-aware `@PreAuthorize` annotation, the `<protect>` declaration takes only a comma-separated list of roles in the `access` attribute (similar to the JSR-250 `@RolesAllowed` annotation).

It is also possible to use a simple wildcard match as part of the stated method name, for example, we might protect all setters of a given bean as follows:

```
<security:intercept-methods>
  <security:protect access="ROLE_USER" method="set*"/>
</security:intercept-methods>
```

Method name matching can be performed by including a leading or trailing regular expression wildcard indicator (*). The presence of such an indicator will perform a wildcard search on the method name, adding the interceptor to any method matching the regular expression. Please note that other common regular expression operators (such as ? or []) are not supported. Please consult the relevant Java documentation for assistance on formulating and understanding basic regular expressions. Any other more complex wildcard or regex matching is not allowed.

It is not common to see this type of security declaration in new code, as more expressive options exist, but it is good to be aware of this type of security decoration so that you can recognize it as an option in your method security toolbelt. This type of method security declaration can be especially useful in cases where adding annotations to relevant interfaces or classes is not an option, such as when you are dealing with securing components in third-party libraries.

Method security rules incorporating method parameters

Logically, writing rules that refer to method parameters in their constraints seems sensible for certain types of operations. For example, it might make sense for us to restrict the changePassword method further, so that a user attempting to invoke the method must satisfy two constraints:

- The user must be attempting to change their own password, or
- The user is an administrator (in which case it is valid for the user to change anyone's password, possibly through an administrative interface)

While it's easy to see how we could alter the rule to restrict the method invocation only to administrators, it's not clear how we would determine if the user is attempting to change their own password.

Fortunately, the SpEL binding used by the Spring Security method annotations supports more sophisticated expressions, including expressions that incorporate method parameters.

```
@PreAuthorize("#username == principal.username and hasRole('ROLE_
USER')")
public void changePassword(String username, String password);
```

You can see here that we've augmented the SpEL directive we used in the first exercise with a check against the username of the principal against the username method parameter (#username — the method parameter name prefixed with a # symbol). The fact that this powerful feature of method parameter binding is available should get your creative juices flowing and allow you to secure method invocations with a very precise set of logical rules.

How method parameter binding works

Using a very similar design to what we saw in Chapter 2 with <intercept-url> authorization expressions, an expression handler—an implementation of the interface o.s.s.access.expression.method.MethodSecurityExpressionHandler—is responsible for setting up the SpEL context under which expressions are evaluated. The MethodSecurityExpressionHandler uses o.s.s.access.expression. method.MethodSecurityExpressionRoot as the expression root, which (much as WebSecurityExpressionRoot did for URL authorization expressions) exposes a set of methods and pseudo-properties to the SpEL expressions for evaluation.

Almost all of the same out of the box expression bits (for example, `hasRole`.) that we saw in Chapter 2 are available in the context of method security, plus one more method involved with access control list checking (covered in *Chapter 7*, *Access Control Lists*) and another pseudo-property that enables powerful role-based filtering of data.

You may note that the `principal` pseudo-property we referenced in the previous example is one of the available expression operands that becomes more helpful in method security expression than in SpEL expressions at the web tier. The `principal` pseudo-property will return the principal placed in the current `Authentication` object, typically either a `String` (username) or `UserDetails` implementation—this means that any `UserDetails` properties or methods could be used to refine method access restrictions.

The following diagram illustrates the functionality in this area:
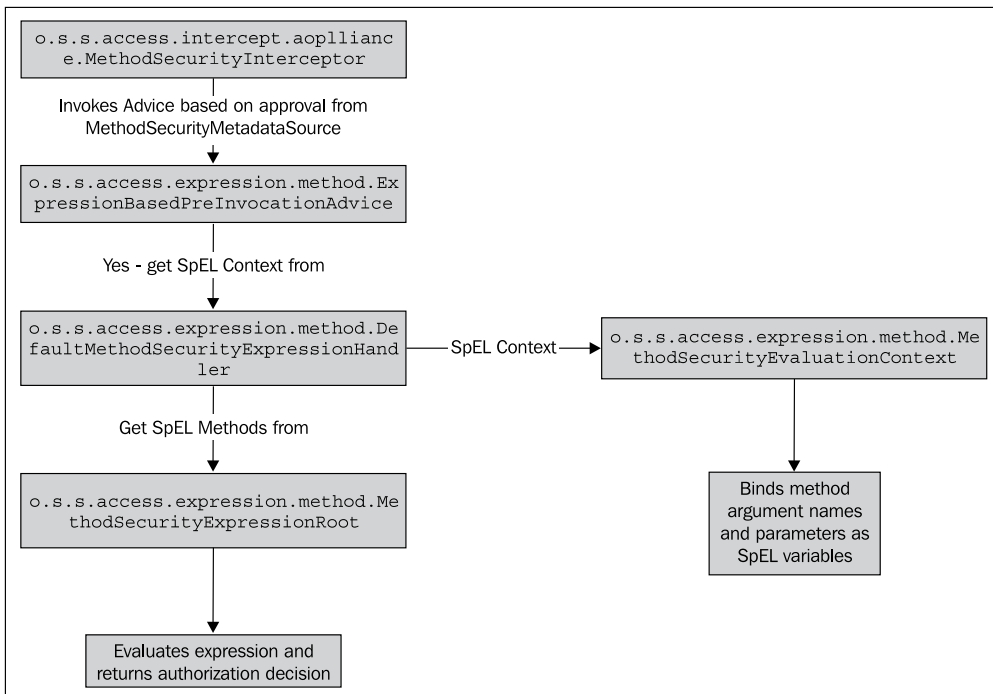
SpEL variables are referenced with the hash (#) prefix. One important note is that in order for method argument names to be available at runtime, debugging symbol table information must be retained after compilation. Common methods of enabling this are listed as follows:

- If you are using the `javac` compiler, you will need to include the `-g` flag when building your classes

- When using the `<javac>` task in ant, add the attribute `debug="true"`
- In Maven, set the property `maven.compiler.debug=on` when building your POM

Consult your compiler, build tool, or IDE documentation for assistance on configuring this same setting in your environment.

Securing method data through Role-based filtering

Two final Spring Security-dependent annotations are `@PreFilter` and `@PostFilter`, which are used to apply security-based filtering rules to `Collections` or `Arrays` (with `@PostFilter` only). This type of functionality is referred to as **security trimming** or **security pruning**, and involves using the principal's security credentials at runtime to selectively remove members from a set of objects. As you might expect, this filtering is performed using SpEL expression notation within the annotation declaration.

We'll work through an example with JBCP Pets where we want to display a special category for users who have an account on the system, called **Customer Appreciation Specials**. Additionally, we'd like to use the `customersOnly` property of the store `Category` object to ensure that certain categories of products are displayed only to customers of the store.

The code involved is straightforward for a Spring MVC web application. The `com.packtpub.springsecurity.web.controller.HomeController` that is used to display the home page of the store, has code to expose categories—a `Collection` of `Category` objects—to the user on the home page:

```
@Controller
public class HomeController extends BaseController {
@Autowired
  private IProductService productService;

  @ModelAttribute("categories")
  public Collection<Category> getCategories() {
    return productService.getCategories();
  }

  @RequestMapping(method=RequestMethod.GET,value="/home.do")
  public void home() {
  }
}
```

The business layer `IProductService` implementation delegates to a data layer `IProductDao` provider. For the purposes of simplicity, the `IProductDao` implementation uses a hard-coded list of `Category` objects.

Adding Role-based data filtering with @PostFilter

Much as we did with method security by authorization, we'll put the `@PostFilter` security filtering directive at the business layer. The syntax in this case will look as follows:

```
@PostFilter("(!filterObject.customersOnly) or (filterObject.
customersOnly and hasRole('ROLE_USER'))")
Collection<Category> getCategories();
```

Before we can understand why this works, we should review the processing flow for the `@PostFilter` annotation:
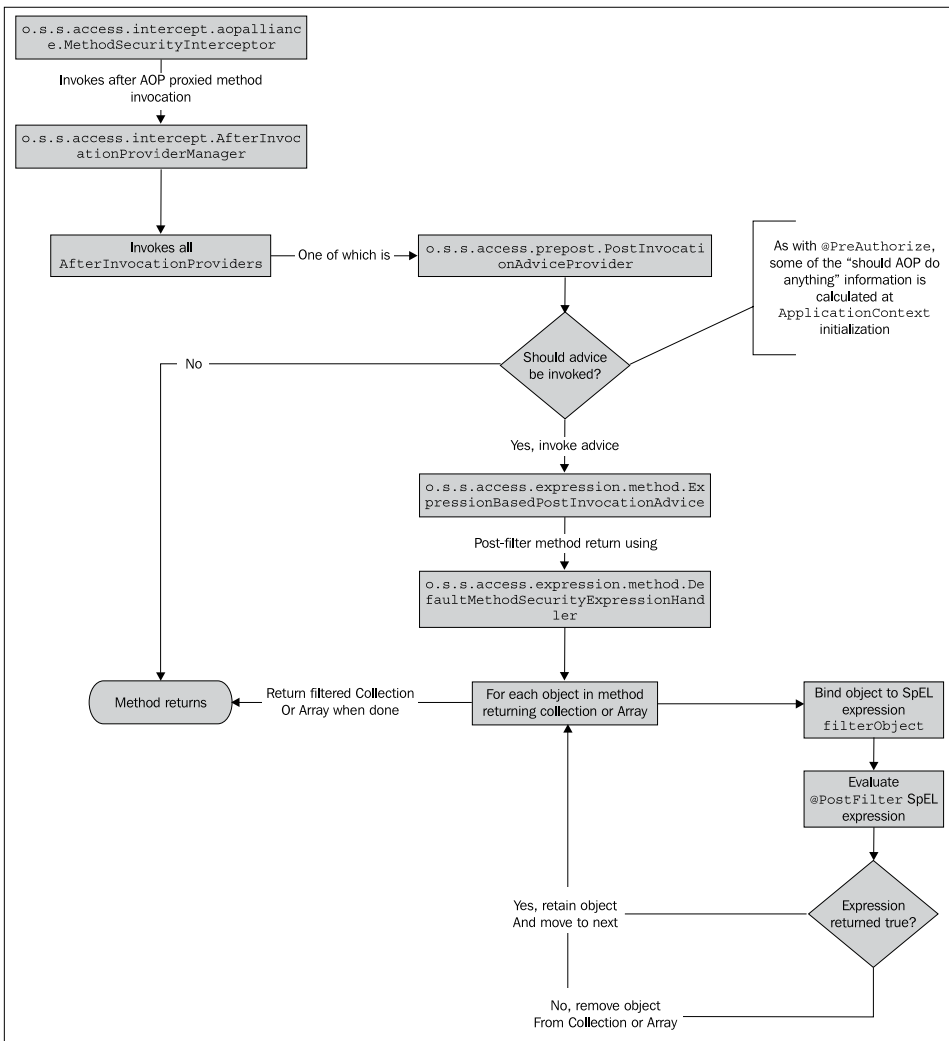
We can see that, again making use of standard Spring AOP constructs, the `o.s.s.access.expression.method.ExpressionBasedPostInvocationAdvice` is executed within an `after` AOP handler, and this advice is used to filter the `Collection` or `Array` returned from the targeted method. As with `@PreAuthorize` annotation processing, the `DefaultMethodSecurityExpressionHandler` is again responsible for building the SpEL evaluation context and evaluation of the expression.

The effect that this change has on the application can be seen when accessing the JBCP Pets home page as a guest, and then as a logged-in user. You can see that when logged in as a registered user, the **Customer Appreciation Specials** category appears for registered users!
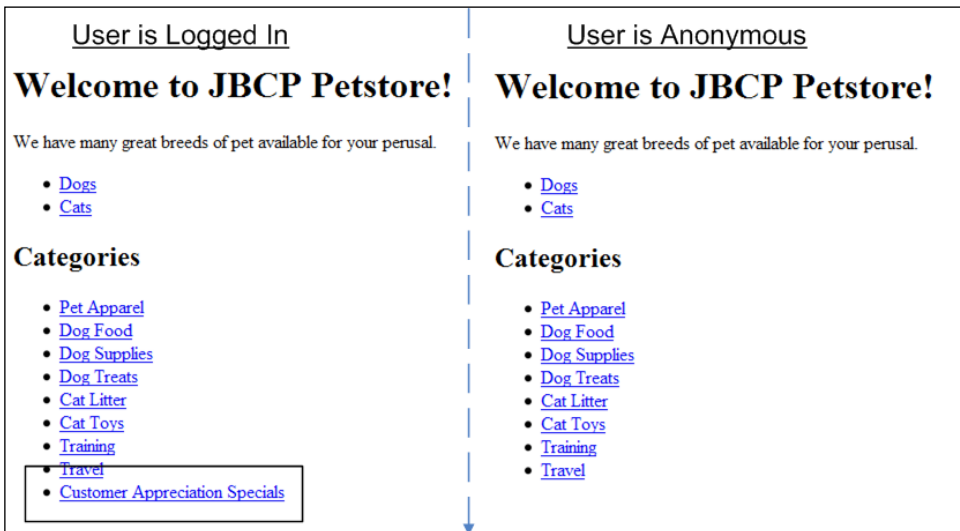
Now that we've reviewed the processing of method post-filtering, let's pull apart the SpEL expression we used to filter categories. For brevity, we refer to `Collection` as the method return value, but be aware that `@PostFilter` works with either `Collection` or `Array` method return types.

- `filterObject` is re-bound to the SpEL context for each element in the `Collection`. This means that if your method is returning a `Collection` with 100 elements, the SpEL expression will be evaluated for each.

- The SpEL expression must return a Boolean value. If the expression evaluates to `true`, the object will remain in the `Collection`, while if the expression evaluates to `false`, the object will be removed.

In most cases, you'll find that `Collection` post-filtering saves you complexity of writing boilerplate code that you would likely be writing anyway.

Take care that you understand how `@PostFilter` works conceptually—unlike `@PreAuthorize`, `@PostFilter` specifies method behavior and not a precondition. Some object oriented purists may argue that `@PostFilter` isn't appropriate for inclusion as a method annotation, and such filtering should instead be handled through code in a method implementation.

> **Safety of Collection filtering**
>
> Be aware that the actual `Collection` returned from your method will be modified! In some cases, this isn't desirable behavior, so you should ensure that your method returns a `Collection` which can be safely modified. This is especially important if the returned `Collection` is an ORM-bound one, as post-filter modifications could inadvertently be persisted to the ORM data store!

Spring Security also offers functionality to pre-filter method parameters which are `Collections`—let's try implementing it now.

Pre-filtering collections with method @PreFilter

The `@PreFilter` annotation can be applied to a method to filter `Collection` elements that are passed into a method based on the current security context. Functionally, once it has a reference to a `Collection`, this annotation behaves exactly the same as the `@PostFilter` annotation, with a couple of exceptions:

- `@PreFilter` supports only `Collection` arguments, and does not support `Array` arguments.

- `@PreFilter` takes an additional, optional attribute `filterTarget`, which is used to specifically identify the method parameter to filter when the annotated method has more than one argument.

As with `@PostFilter`, keep in mind that the original `Collection` passed to the method is permanently modified. This may not be desirable behavior, so ensure that callers know that the `Collection` may be security trimmed after the method is invoked!

To explore use of this method of filtering, we'll temporarily adjust the `getCategories` method that we applied the `@PostFilter` annotation to, by having it delegate its filtering to a new method. Modify the `getCategories` method as follows:

```
@Override
public Collection<Category> getCategories() {
   Collection<Category> unfilteredCategories = productDao.
getCategories();
   return productDao.filterCategories(unfilteredCategories);
}
```

We'll need to add the `filterCategories` method to the `IProductDao` interface and implementation. The `@PreFilter` annotation will be added to the interface declaration, as follows:

```
@PreFilter("(!filterObject.customersOnly) or (filterObject.
customersOnly and hasRole('ROLE_USER'))")
public Collection<Category> filterCategories(Collection<Category>
categories);
```

Once you have the method and `@PreFilter` annotation declared on the interface, simply add an empty implementation (although you could perform further filtering within the method if business requirements dictate). Add the following method body to `ProductDao`:

```
@Override
public Collection<Category> filterCategories(Collection<Category>
categories) {
   return categories;
}
```

At this point, you can verify that the functionality works properly by removing the `@PostFilter` annotation from the `IProductService` interface, and you'll see that the behavior is the same as before.

Why use a @PreFilter at all?

At this point, you may be scratching your head at why a `@PreFilter` is useful at all, given that `@PostFilter` functions just the same and makes more logical sense on methods that return data.

`@PreFilter` does in fact have a number of uses, some of which overlap with `@PostFilter`, but remember that when declaring security restrictions, it's OK to be redundant—it's better to be overly cautious than to potentially open yourself up to a vulnerability.

The following are some situations where @PreFilter could be useful:

- Many applications have a facility at the data layer to execute queries with a series of parameters. @PreFilter could be used to ensure security scrubbing of parameters passed into database queries.

- There may be cases when the business tier will be assembling information to return from many data sources. Input to each of the data sources can be properly security trimmed so that the user doesn't inadvertently get search results or data that they shouldn't have access to.

- @PreFilter could be used to do location-or relationship-based filtering—for example, it could form the basis of implicit search criteria based on categories the user has clicked on, or items they have ordered.

Hopefully this provides you with some ideas on where pre-and post-filtering of Collections can add an extra layer of security to your application!

A fair warning about method security

Take note of this very important warning about implementing and understanding method security—in order to truly excel at implementation of this powerful type of security, it is critical that you comprehend how it works behind the scenes. Lack of understanding of AOP, at a conceptual and tactical level, is the number one cause of frustration among users of method security. Make sure you read thoroughly, not only this chapter, but also *Chapter 7* of the *Spring 3 Reference Documentation, Aspect Oriented Programming with Spring*.

Prior to implementing method security on an existing application, it's a good idea to also review the application's compliance to object-oriented design principles. You will have fewer unexpected errors when implementing method security if your application already makes proper use of interfaces and encapsulation.

Summary

In this chapter, we have covered most of the remaining areas in standard Spring Security implementations which deal with authorization. We've learned enough to take a thorough pass through the JBCP Pets online store and verify that proper authorization checks are in place at all tiers of the application, to ensure that malicious users cannot manipulate or access data to which they do not have access.

Specifically, we:

- Learned about planning authorization, and user/group mapping in our application design process
- Developed two techniques for micro-authorization, filtering out in-page content based on authorization or other security criteria using the Spring Security JSP tag library and Spring MVC controller data binding
- Explored several methods of securing business functions and data in the business tier of our application and supporting a rich declarative security model which is tightly integrated with the code

At this point, we've wrapped up coverage of much of the important Spring Security functionality that you're likely to encounter in most standard secure web application development scenarios.

If you've read this far without stopping for a breather, now is a good time to take a break, review what we've learned to this point, and spend some time exploring the sample code and the Spring Security code itself.

In the following two chapters, we'll cover several advanced customization and extension scenarios, as well as the access control list (domain object model) module of Spring Security. Exciting topics, for sure!

6
Advanced Configuration and Extension

Up to now, we've covered a lot of theory, along with the architecture and usage of the majority of Spring Security components. Our JBCP Pets commerce site is well on its way to being a model citizen of the secured web, and we're ready to dig into some difficult challenges.

During the course of this chapter, we'll:

- Implement our own security filter, approaching an interesting problem of augmenting site security through the use of selective IP filtering by user role
- Build a custom `AuthenticationProvider` and the required supporting classes
- Understand and implement anti-hacker measures known as session fixation protection and concurrent session control
- Utilize functionality included in concurrent session control to build some simple user session reporting enhancements
- Configure, and then customize, access denied behavior and exception handling
- Build a Spring bean-based configuration of Spring Security, eschewing the convenience of the security namespace's `<http>` configuration style, and directly wiring and instantiating the entire Spring Security stack from the ground up
- Review how to configure session handling and creation with a Spring bean-based security configuration
- Examine the pros and cons of the `<http>` configuration style versus the Spring bean-based configuration

- Learn about the `AuthenticationEvent` architecture along with available event handlers and customization
- Build a customized SpEL expression voter, and create a new SpEL method to be used in `<intercept-url>` expressions

Writing a custom security filter

A common customization scenario for secured applications is the use of custom servlet filters, which can function as additional layers of application-specific security, enhance the user's experience by providing complementary information, and remove potentially malicious behavior.

IP filtering at the servlet filter level

One useful enhancement that would please the JBCP Pets security auditors would be further tightening of restrictions around use of the administrative account, or (even better) of any user with administrative access to the site.

We'll tackle this problem with a filter that ensures that users with the `ROLE_ADMIN` role can access the system only from a specified set of IP addresses. We'll do simple address matching here, but you could easily extend this example to apply IP masking, read the IP addresses from a database, and so on.

Studious readers will note that there are several other methods of achieving this functionality, including more sophisticated SpEL `<intercept-url>` access declarations; however, for the purpose of illustration, this is a straightforward example to follow. Keep in mind that in the real world, things such as **Network Address Translation** (**NAT**) and dynamic IP addresses can make IP-based rules on public, unmanaged networks fragile.

Writing our custom servlet filter

Our custom filter will be configured to take a target role along with a list of IP addresses for which the role will be allowed. We've called this class `com.packtpub.springsecurity.security.IPRoleAuthenticationFilter`, and defined it as follows. This code is a bit complex, so we'll omit some of the less important bits in the code listing. Please consult the code for this chapter for the full source code of the class.

```
package com.packtpub.springsecurity.security;
// imports omitted
public class IPRoleAuthenticationFilter extends OncePerRequestFilter
  {}
```

We can see that our filter extends the `o.s.web.filter.OncePerRequestFilter` base class from the Spring web library. While this isn't strictly required, it's handy for filters with more complex configurations, intended only to be run once, so we'd recommend it.

```
private String targetRole;
private List<String> allowedIPAddresses;
```

Our filter will have two properties—the target role (for example, ROLE_ADMIN), and a `List` of allowed IP addresses. These will be configured through standard Spring bean definition, as we'll see shortly. Finally, we get to the meat of the bean, which is the `doFilterInternal` method.

```
@Override
public void doFilterInternal(HttpServletRequest req,
HttpServletResponse res, FilterChain chain)
throws IOException, ServletException {
    // before we allow the request to proceed, we'll first get the
user's role
    // and see if it's an administrator
    final Authentication authentication = SecurityContextHolder.
getContext().getAuthentication();
    if (authentication != null && targetRole != null) {
        boolean shouldCheck = false;
    // look if the user is the target role
        for (GrantedAuthority authority : authentication.
getAuthorities()) {
            if(authority.getAuthority().equals(targetRole)) {
                shouldCheck = true;
                break;
                 }
        }
        // if we should check IP, then check
        if(shouldCheck && allowedIPAddresses.size() > 0) {
            boolean shouldAllow = false;
            for (String ipAddress : allowedIPAddresses) {
                if(req.getRemoteAddr().equals(ipAddress)) {
                    shouldAllow = true;
                    break;
                }
            }

            if(!shouldAllow) {
                // fail the request
```

```
                  throw new AccessDeniedException("Access has been
            denied for your IP address: "+req.getRemoteAddr());
            }
         }
      } else {
         logger.warn("The IPRoleAuthenticationFilter should be placed
   after the user has been authenticated in the filter chain.");
      }
      chain.doFilter(req, res);
   }
   // accessors (getters and setters) omitted
   }
```

As you can see, this code is very straightforward, and uses the `SecurityContext` to acquire the `Authentication` information about the current secured request, just as we've done in exercises from prior chapters. You may note that there isn't a whole lot here that's specific to Spring Security, besides the method of acquiring the user's role (`GrantedAuthority`) and the use of `AccessDeniedException`, to indicate that access is denied.

Let's work through the process of configuring our custom filter as a Spring bean.

Configuring the IP servlet filter

Configuration of the filter is a simple Spring bean. We can configure it in the `dogstore-base.xml` file.

```
<bean id="ipFilter" class="com.packtpub.springsecurity
   .security.IPRoleAuthenticationFilter">
   <property name="targetRole" value="ROLE_ADMIN"/>
   <property name="allowedIPAddresses">
     <list>
       <value>1.2.3.4</value>
     </list>
   </property>
</bean>
```

Using standard Spring bean configuration syntax, we can supply a list of IP address values. In this case, `1.2.3.4` is obviously not a correct IP address (`127.0.0.1` would be a good one if you are doing local development), but it provides us with a convenient way to test if the filter works!

Finally, we'll add the filter to the Spring Security filter chain.

Adding the IP servlet filter to the Spring Security filter chain

Servlet filters that are intended to be interjected in the midst of the Spring Security filter chain are always positioned relative to other filters, which already exist in the filter chain. Custom filter chain members are configured in the `<http>` element, with a simple bean reference and a positioning indicator, as we'll see when we configure the IP servlet filter:

```
<http>
  <custom-filter ref="ipFilter"
  before="FILTER_SECURITY_INTERCEPTOR"/>
</http>
```

Remember that our filter relies on the `SecurityContext` and `Authentication` objects being available and populated, in order to verify the user's `GrantedAuthority`. As such, we'll want to position this filter right before the `FilterSecurityInterceptor`, which (as you may recall from *Chapter 2*, *Getting Started with Spring Security*) is responsible for determining whether the user has provided correct credentials to give them access to the system. At this point in the servlet chain, the user's identity is well known, so this is the appropriate location for the insertion of our filter.

You can restart the application now, and see that your login as the `admin` user will be restricted due to the new IP filter. Play around with it a little until you really understand how all the pieces fit together!

> **Extending the IP filter**
>
> For a more complex set of requirements, it may make sense to extend this filter to allow more sophisticated matching on roles or IP addresses (subnet matching, IP ranges, and so on). Unfortunately, no widely available library exists in Java to perform these kinds of calculations--which is quite surprising, considering the ubiquity of IP filtering in secured environments.

Try enhancing the IP filter to include functionality that we haven't described here—getting your hands dirty is the best method of learning, and adapting exercises to real-world examples is a great way to make an abstract concept feel more concrete!

Writing a custom AuthenticationProvider

In many scenarios where your application requirements fall outside the bounds of standard Spring Security functionality, you'll need to implement your own `AuthenticationProvider`. Recollect from Chapter 2 that the role of the `AuthenticationProvider`, in the overall authentication process, is to accept presented credentials (known as an `Authentication` object or **authentication token**) from a principal's request and verify their correctness and validity.

Implementing simple single sign-on with an AuthenticationProvider

Usually, an application will allow for one or more capabilities for users or agents to sign on. However, it's also quite common, especially in widely available applications, to allow for multiple methods of sign-on for different users of the system.

Let's assume that our system is integrating with a simple "single sign-on" provider, where the username and password are sent in the HTTP request headers `j_username` and `j_password`, respectively. Additionally, there's a `j_signature` header that contains an encoded string with the username concatenated with the password using an arbitrary algorithm, to further assist in request security.

> Please do not use this sample as the basis of a real single sign-on solution! It is contrived, and intended only to illustrate the steps required to implement a fully custom `AuthenticationProvider`. True SSO solutions are significantly more secure and involve multiple handshakes to establish trustworthiness of credentials. Spring Security offers support for several SSO solutions, including **Central Authentication Service** (**CAS**) and SiteMinder, which we'll cover in *Chapter 10, Single Sign On with Central Authentication Service*. In fact, Spring Security provides a similar filter used for SiteMinder request header-based authentication, the `o.s.s.web.authentication.preauth.RequestHeaderAuthenticationFilter`, which is also a good example of this type of functionality.

Our algorithm would expect the following data values in the headers, for a login by the `admin` user:

Request Header	Value
j_username	admin
j_password	admin
j_signature	admin|+|admin

Normally, an `AuthenticationProvider` will look for a particular specialization of `AuthenticationToken` that has been populated by a servlet filter earlier in the chain (earlier than the `AuthenticationManager` access checking is done, specifically), as illustrated in the following diagram:
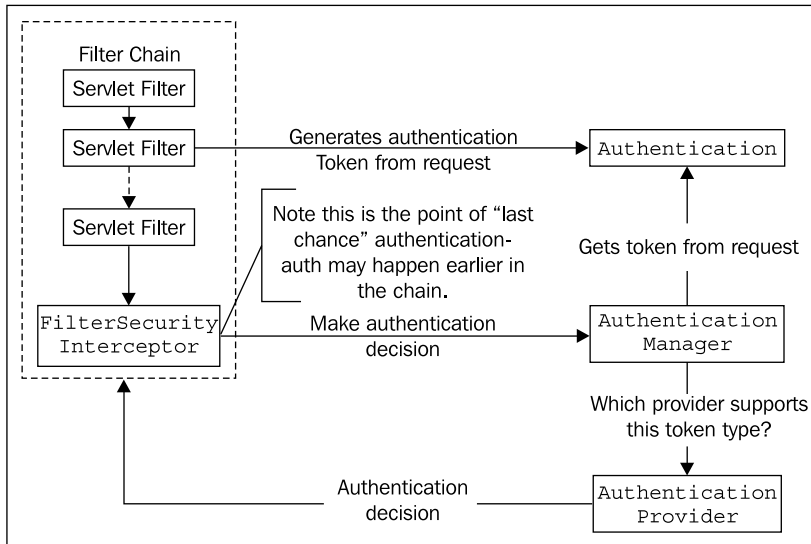
In this way there is a bit of a disconnected relationship between the supplier of the `AuthenticationToken` and the consuming `AuthenticationProvider`. So, when you are implementing a custom `AuthenticationProvider`, it's quite likely you'll be implementing a custom servlet filter as well, whose responsibility is to supply a specialized `AuthenticationToken`.

Customizing the authentication token

Our approach for this customization will be to utilize as much of Spring Security's baseline functionality as possible. As such, we'll extend and augment base classes like the `UsernamePasswordAuthenticationToken`, to add the new field to store our encoded signature string. The resulting class, `com.packtpub.springsecurity. security.SignedUsernamePasswordAuthenticationToken`, is as follows:

```
package com.packtpub.springsecurity.security;
// imports omitted
  public class SignedUsernamePasswordAuthenticationToken
    extends UsernamePasswordAuthenticationToken {
```

```
    private String requestSignature;
    private static final long serialVersionUID =
    3145548673810647886L;
  /**
   * Construct a new token instance with the given principal,
credentials, and signature.
   *
   * @param principal the principal to use
   * @param credentials the credentials to use
   * @param signature the signature to use
   */
  public SignedUsernamePasswordAuthenticationToken(String principal,
    String credentials, String signature) {
    super(principal, credentials);
    this.requestSignature = signature;
    }
  public void setRequestSignature(String requestSignature) {
    this.requestSignature = requestSignature;
    }
  public String getRequestSignature() {
    return requestSignature;
    }
}
```

We can see that this `SignedUsernamePasswordAuthenticationToken` is a very
simple POJO extension of the `UsernamePasswordAuthenticationToken`. Tokens
need not be very complex—their sole purpose is to encapsulate presented
credentials for later verification.

Writing the request header processing servlet filter

Now, we'll write the code for the servlet filter that is responsible for translating
the request headers into our newly minted token. Again, we'll extend from the
appropriate Spring Security base class. In this case, `o.s.s.web.authentication.`
`AbstractAuthenticationProcessingFilter` fits our needs.

> The base filter, `AbstractAuthenticationProcessingFilter`, is
> a common superclass of many of the authentication-oriented filters in
> Spring Security (which are OpenID, Central Authentication Service,
> and form-based username and password login). This class provides
> standardized authentication logic and wires other important resources
> such as `RememberMeServices` and `ApplicationEventPublisher`
> (covered later in this chapter) properly.

Let's review the code now:

```
package com.packtpub.springsecurity.security;
// imports omitted
  public class RequestHeaderProcessingFilter extends
    AbstractAuthenticationProcessingFilter {
  private String usernameHeader = "j_username";
  private String passwordHeader = "j_password";
  private String signatureHeader = "j_signature";
  protected RequestHeaderProcessingFilter() {
    super("/j_spring_security_filter");
    }
  @Override
  public Authentication attemptAuthentication
    (HttpServletRequest request,HttpServletResponse response)
    throws AuthenticationException,
      IOException, ServletException {
    String username = request.getHeader(usernameHeader);
    String password = request.getHeader(passwordHeader);
    String signature = request.getHeader(signatureHeader);
    SignedUsernamePasswordAuthenticationToken authRequest =
      new SignedUsernamePasswordAuthenticationToken
      (username, password, signature);
    return this.getAuthenticationManager().authenticate(authRequest);
    }
  // getters and setters omitted below
}
```

We can see that our simple filter looks for three named headers, as we had planned (configurable through bean properties, if needed), and listens by default on the URL /j_spring_security_filter. Just as with other Spring Security filters, this is a virtual URL that is recognized by our filter's base class, AbstractAuthenticationProcessingFilter, upon which action is taken by the filter in an attempt to create an Authentication token and then authenticate the user's request.

> **Differentiating components participating in authentication token workflow**
>
> It's easy to become confused by the terminology, interface, and class names in this functional area. The interface representing authentication token classes is o.s.s.core.Authentication, and implementations of this interface end with the suffix AuthenticationToken. This is an easy way to identify the authentication implementation classes that are shipped with Spring Security!

For the purposes of this example, we've kept the error checking to a minimum here. Presumably, the application would be verifying whether or not all expected headers were supplied, and reporting an exception or redirecting the user if things weren't found to be as expected.

A minor configuration change is required to insert our filter into the filter chain.

```
<http auto-config="true" ...>
  <custom-filter ref="requestHeaderFilter"
  before="FORM_LOGIN_FILTER"/>
</http>
```

You can see that the final request in the filter code is to the `AuthenticationManager`, to perform authentication. This will delegate (eventually) to the configured `AuthenticationProviders`, one of which would be expected to support checking on a `SignedUsernamePasswordAuthenticationToken`. Next, we'll need to write an `AuthenticationProvider` responsible for doing just this.

Writing the request header AuthenticationProvider

Now, we'll write an `AuthenticationProvider` implementation, `com.packtpub.springsecurity.security.SignedUsernamePasswordAuthenticationProvider`, responsible for validating the signature in our custom `Authentication` token.

```
package com.packtpub.springsecurity.security;
// imports omitted
  public class SignedUsernamePasswordAuthenticationProvider
  extends DaoAuthenticationProvider {
    @Override
    public boolean supports(Class<? extends Object> authentication) {
      return (SignedUsernamePasswordAuthenticationToken
      .class.isAssignableFrom(authentication));
    }

  @Override
  protected void additionalAuthenticationChecks
  (UserDetails userDetails,
    UsernamePasswordAuthenticationToken authentication)
    throws AuthenticationException {
    super.additionalAuthenticationChecks
    (userDetails, authentication);

    SignedUsernamePasswordAuthenticationToken signedToken =
      (SignedUsernamePasswordAuthenticationToken) authentication;

    if(signedToken.getRequestSignature() == null) {
      throw new BadCredentialsException(messages.getMessage(
      "SignedUsernamePasswordAuthenticationProvider
```

```
        .missingSignature", "Missing request signature"),
        isIncludeDetailsObject() ? userDetails : null);
      }
  // calculate expected signature
    if(!signedToken.getRequestSignature()
    .equals(calculateExpectedSignature(signedToken))) {
    throw new BadCredentialsException(messages.getMessage
    ("SignedUsernamePasswordAuthenticationProvider
    .badSignature", "Invalid request signature"),
    isIncludeDetailsObject() ? userDetails : null);
    }
  }

  private String calculateExpectedSignature
    (SignedUsernamePasswordAuthenticationToken signedToken) {
      return signedToken.getPrincipal() + "|+|" +
      signedToken.getCredentials();
    }
  }
```

You can see that we've extended a framework class, the
`DaoAuthenticationProvider` again, because the useful data access code there
is still required for the actual verification of the user's password, and to load
`UserDetails` from the `UserDetailsService`.

This class is a little more complicated, so we'll go through the methods one at a time.

The `supports` method, overridden from the superclass, indicates to the
`AuthenticationManager` the expected runtime type of the `Authentication` token
that this `AuthenticationProvider` has an interest in evaluating.

Next, the `additionalAuthenticationChecks` method is invoked by the superclass
to allow subclasses to perform any additional subclass-specific validation of the
token. This fits in perfectly with our strategy, so we simply add our new checks on
the signature of the supplied token. We're almost done with our custom "simple
SSO" implementation, so only one configuration step remains.

Combining AuthenticationProviders

A common use is the need to combine one or more `AuthenticationProvider`
interfaces, as the user may log into the application with one of the several methods
of authentication.

While we haven't covered any other `AuthenticationProvider` up to this point, we can assume the following desired workflow using the standard username or password form-based authentication, and the custom simple SSO provider that we implemented in the previous exercise. When combining multiple `AuthenticationProvider` interfaces, each `AuthenticationProvider` will examine the `AuthenticationToken` supplied by the servlet filters in the filter chain, and handle the token only if the token type is supported. In this way, it's not generally harmful to allow your application to simultaneously support different methods of authentication within a single configuration stack.

Combining `AuthenticationProviders` is actually trivial. We simply need to declare another `authentication-provider` reference in our `dogstore-security.xml` configuration file.

```
<authentication-manager alias="authenticationManager">
  <authentication-provider ref=
  "signedRequestAuthenticationProvider"/>
  <authentication-provider user-service-ref="jdbcUserService">
    <password-encoder ref="passwordEncoder" >
      <salt-source ref="saltSource"/>
    </password-encoder>
  </authentication-provider>
</authentication-manager>
```

As with many of the other Spring bean references in our security configuration file, the `signedRequestAuthenticationProvider` reference will resolve to our `AuthenticationProvider`, which we'll configure along with our other Spring beans in `dogstore-base.xml`.

```
<bean id="signedRequestAuthenticationProvider"
  class="com.packtpub.springsecurity.security
  .SignedUsernamePasswordAuthenticationProvider">
  <property name="passwordEncoder" ref="passwordEncoder"/>
  <property name="saltSource" ref="saltSource"/>
  <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

The bean properties on our custom `AuthenticationProvider` are actually required by the superclass. They'll simply refer to the same bean declarations that we've referred to in the second `authentication-provider` declared along with the `AuthenticationManager`.

We've finally finished with the coding and configuration required to support this single sign-on feature, so give yourself a round of applause! However, one small problem remains—how can we manipulate the request headers to simulate our SSO provider?

# Simulating single sign-on with request headers

Although our scenario is contrived, there are several commercial and open source single sign-on solutions that can be configured to send credentials using HTTP request headers; most notably CA (formerly Netegrity) SiteMinder.

> It is important to note that applications integrated behind SSO solutions are expected to be inaccessible by direct user requests. It is almost always the case that the SSO provider functions as a proxy through which user requests flow (and are secured) or that the provider retains all knowledge of passwords and isolates the individual secured applications from this knowledge.
>
> Do not deploy an SSO-enabled application without a full understanding of the hardware, network, and security infrastructure it is going into!

A browser extension for Mozilla Firefox, called Modify Headers (available from `http://modifyheaders.mozdev.org/`), is very straightforward tool for simulating requests that rely on manipulation on HTTP request headers. The following screenshot illustrates how we could use the tool to add the headers that our SSO solution is expecting:
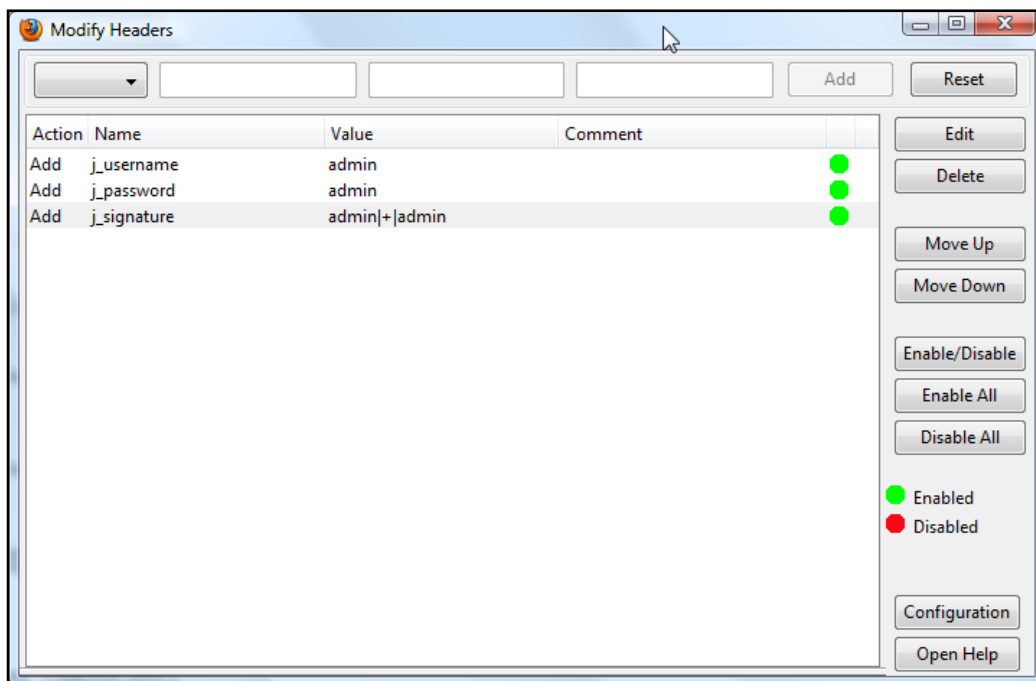
With all the headers marked as **Enabled**, we can simply visit the
URL `http://localhost:8080/JBCPPets/j_spring_security_filter` and
we will see that we're automatically logged into the system! You may also note
that our form-based login continues to work as well, due to our retention of both
`AuthenticationProvider` implementations and corresponding filters in the
filter stack.

Considerations when writing a custom AuthenticationProvider

Although the example we have just seen was probably not illustrative of the type of
`AuthenticationProvider` you are likely to build, the series of steps required
for any custom `AuthenticationProvider` will be very similar. The important
takeaways from this exercise are:

- The responsibility for populating an `Authentication` implementation from
  the user's request is usually placed on a member of the servlet filter chain.
  Depending on whether or not the data needs to validate the credentials
  presented, the authentication component may be extended.

- The responsibility for authenticating a user based on a valid `Authentication`
  token should come from an `AuthenticationProvider` implementation.
  Refer to our discussion of how `AuthenticationProviders` are expected
  to behave, from Chapter 2, for more details.

- In special cases, when an unauthenticated session is detected, a custom
  `AuthenticationEntryPoint` may be required. We will read more about
  this interface later in this chapter, and will also see some examples of a
  custom `AuthenticationEntryPoint` in action when we review Central
  Authentication Service (CAS) integration in Chapter 10.

If you keep these roles in mind when developing an `AuthenticationProvider`
specific to your application's needs, you'll have a lot less confusion at the time of
implementation and debugging!

Session management and concurrency

A common configuration for Spring Security is detection of the same user logging
into the secured application from different active sessions. This is known as
concurrency control, and is part of a class of configurable functionality around
session management. Although this functionality is not, strictly speaking, advanced
configuration, it tends to be quite confusing for new users, and is best picked up once
you have some familiarity with the overall operation of the Spring Security framework.

Session management in Spring Security can be configured in two different ways—session fixation protection and concurrency control. As some aspects of concurrency control build upon the framework established by session fixation protection, we'll examine session fixation first.

Configuring session fixation protection

As we are using the security namespace style of configuration, session fixation protection is already configured on our behalf. If we wanted to explicitly configure it to mirror the default settings, we would do the following:

```
<http auto-config="true" use-expressions="true">
  <!-- ... -->
  <session-management session-fixation-protection="migrateSession"/>
</http>
```

Session fixation protection is a feature of the framework that you most likely won't even notice unless you try to act as a malicious user. We'll show you how to simulate a session stealing attack, but before we do, it's important to understand what session fixation does and the type of attack it prevents.

Understanding session fixation attacks

Session fixation is a type of attack whereby a malicious user attempts to steal the session of an unauthenticated user of your system. This can be done by using a variety of techniques that result in the attacker obtaining the unique session identifier of the user (for example, JSESSIONID). If the attacker creates a cookie or URL parameter with the user's JSESSIONID in it, they can access the user's session.

Although this is obviously a problem, typically, if a user is unauthenticated, they haven't entered any sensitive information (assuming the site security is planned correctly!). This becomes a more critical problem if the same session identifier continues to be used after a user has authenticated. If the same identifier is used after authentication, the attacker may now gain access to the authenticated user's session without ever having to know their username or password!

At this point, you may scoff in disbelief and think this is extremely unlikely to happen in the real world. In fact, session stealing attacks happen frequently. We would suggest (as we did in Chapter 3) that you spend some time reading the very informative articles and case studies on the subject published by the OWASP organization (`http://www.owasp.org/`). Attackers and malicious users are real, and they can do very real damage to your users, your application, or your company, if you don't understand the techniques that they commonly use and know how to avoid them.

The following diagram illustrates how a session fixation attack works:
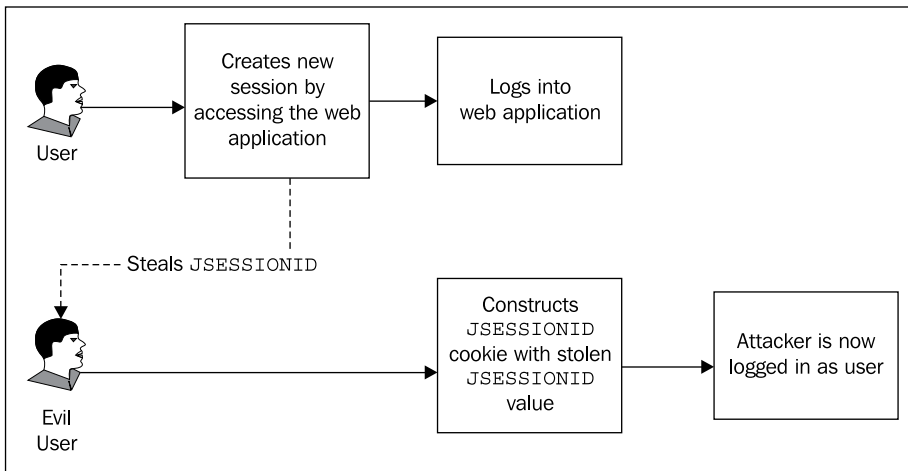
Now that we have seen how an attack like this works, we'll see what Spring Security can do to prevent it.

Preventing session fixation attacks with Spring Security

If we can prevent the same session that the user had prior to authentication from being used after authentication, we can effectively render the attacker's knowledge of the session ID useless. Spring Security session fixation protection solves this problem by explicitly creating a new session when a user is authenticated, and invalidating their old session.

Let's see the following screenshot:
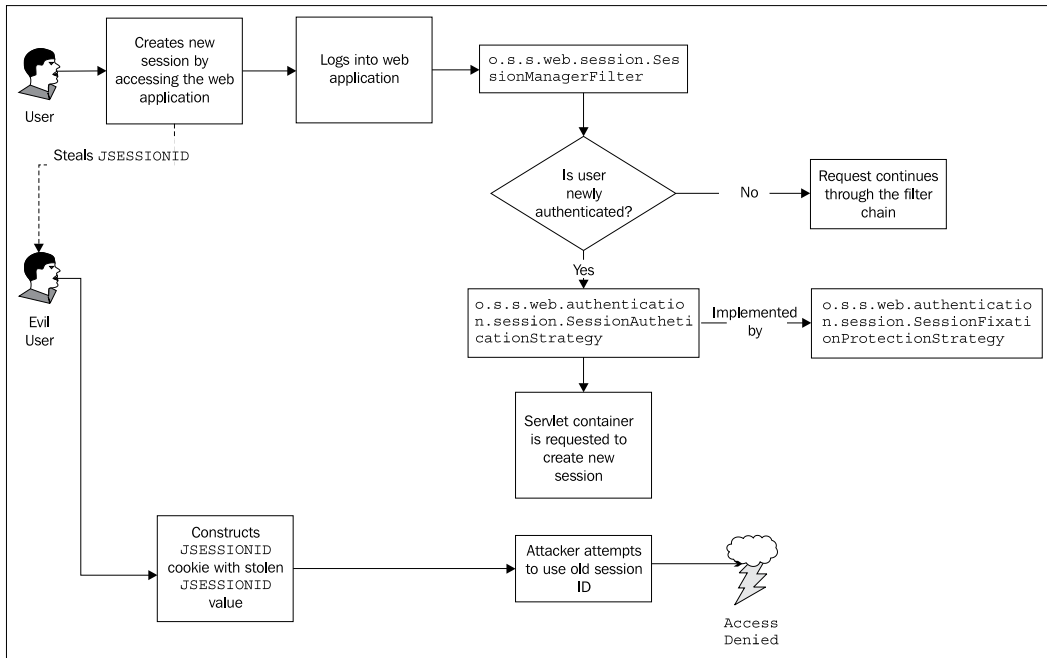
We can see that a new filter, the `o.s.s.web.session.SessionManagementFilter`, is responsible for evaluating if a particular user is newly authenticated. If the user is newly authenticated, a configured `o.s.s.web.authentication.session.SessionAuthenticationStrategy` determines what to do. The `o.s.s.web.authentication.session.SessionFixationProtectionStrategy` will create a new session (if the user already had one), and copy the contents of the existing session to the new one. That's pretty much it—seems simple; however, as we can see in the diagram, it effectively prevents the evil user from reusing the session ID after the unknowing user has authenticated.

Simulating a session fixation attack
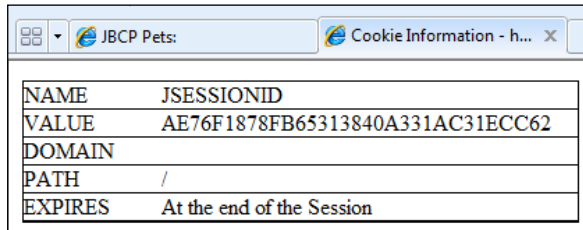
At this point you may want to see what's involved in simulating a session fixation attack. To do this, you'll first need to disable session fixation protection in `dogstore-security.xml`.

```
<session-management session-fixation-protection="none"/>
```

Next, you'll need to open two browsers. We'll initiate the session in Internet Explorer, steal it from there, and our attacker will log in using the stolen session in Firefox. We will use the Internet Explorer Developer Tools (built into IE 8) and the Firefox Web Developer Add-On (refer to Chapter 3 for the URL) in order to view and manipulate cookies.

Open the JBCP Pets home page in Internet Explorer. Next, open the Developer Tools (look in the **Tools** drop-down menu, or hit *F12*) and select **View Cookie Information** from the **Cache** menu. Find the cookie for JSESSIONID with the appropriate domain (this will be blank if you are using localhost).

Copy the session cookie value to the clipboard, and then log in to the JBCP Pets site. If you repeat the **View Cookie Information** command, you'll see that the JSESSIONID did not change after you logged in, making you vulnerable to a session fixation attack!

In Firefox, open the JBCP Pets website. You will have been assigned a session cookie, which we can view using the **Cookies** menu, and the **View Cookie Information** menu option.
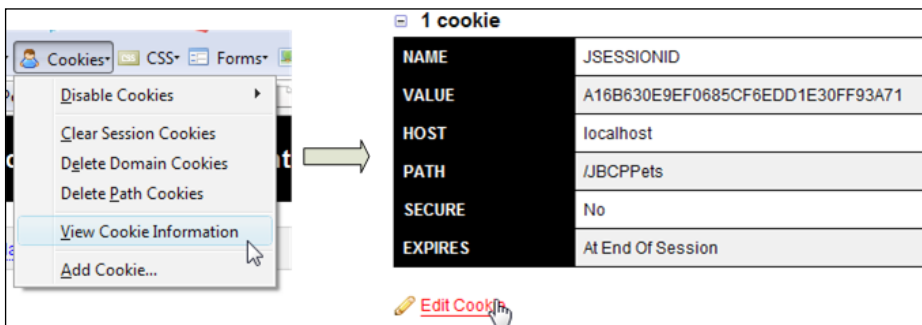
To complete our hack, we'll click the **Edit Cookie** option, and paste in the JSESSIONID cookie that we copied to the clipboard from Internet Explorer as shown in the following screenshot:
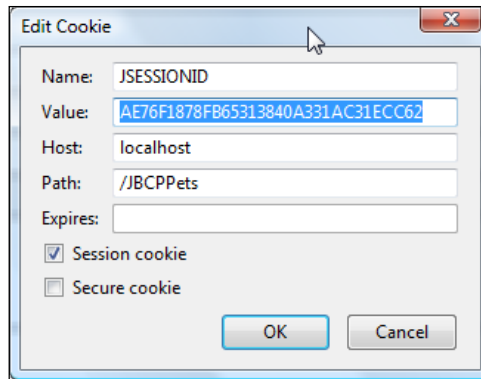
Our session fixation hack is complete! If you now reload the page in Firefox, you will see that you are logged in as the same user that was logged in using Internet Explorer, without requiring knowledge of the username and password. Are you scared of malicious users yet?

Now, re-enable session fixation protection and try this exercise again. You'll see that, in this case, the JSESSIONID changes after the user logs in. Based on our understanding of how session fixation attacks occur, this means that we have reduced the likelihood of an unsuspecting user falling victim to this type of attack. Excellent job!

Cautious developers should take note that there are many methods of stealing session cookies, some of which such as **Cross-Site Scripting** (**XSS**) may make even session fixation-protected sites vulnerable. Please consult the OWASP site for additional resources on preventing these types of attacks.

Comparing session-fixation-protection options

The session-fixation-protection attribute has three options that allow you to alter its behavior.

Attribute value	Description
none	Disables session fixation protection, and (unless other <session-management> attributes are non-default) does not configure a SessionManagementFilter.
migrateSession	When the user is authenticated and a new session is allocated, it ensures that all attributes of the old session are moved to the new session. We'll see that this behavior is configurable when using bean-based configuration, later in this chapter.
newSession	When the user is authenticated, a new session is created and no attributes from the old (unauthenticated) session will be migrated.

In most cases, the default behavior of `migrateSession` will be appropriate for sites that wish to retain important attributes of the user's session (such as click interest, shopping carts, and so on) after the user has been authenticated.

Enhancing user protection with concurrent session control

An enhancement to user security which naturally follows from session fixation protection, is concurrent session control. As described previously, concurrent session control ensures that a single user cannot have more than a fixed number of active sessions simultaneously (typically one). Ensuring that this maximum limit is enforced involves several components working in tandem to accurately track changes in user session activity.

Let's configure the feature, review how it works, and then test it out!

Configuring concurrent session control

Now that we have understood the different components involved in concurrent session control, setting it up should make much more sense. First, we need to enable the `ConcurrentSessionFilter` and related bits in `dogstore-security.xml`.

```
<http auto-config="true" use-expressions="true">
<!-- ... -->
  <session-management>
    <concurrency-control max-sessions="1"/>
  </session-management>
</http>
```

Next, we need to enable the `o.s.s.web.session.HttpSessionEventPublisher` in the `web.xml` deployment descriptor, so that the servlet container will notify Spring Security (through the `HttpSessionEventPublisher`) of session lifecycle events.

```
<listener>
  <listener-class>
    org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<listener>
  <listener-class>
    org.springframework.security.web.session
    .HttpSessionEventPublisher
  </listener-class>
</listener>
<servlet>
  <servlet-name>dogstore</servlet-name>
```

With these two configuration bits in place, concurrent session control will now be activated. Let's see what it actually does, and then we'll review some steps to reproduce its protection over our users' sessions.

Understanding concurrent session control

We mentioned previously that concurrent session control attempts to limit access by the same user through different sessions. Based on our understanding of session stealing-type attacks, we can see that concurrent session control can mitigate the possibility that an attacker could use a stolen session while the original, unsuspecting user, is logged in. Why do you think that is so?

Concurrent session control uses an `o.s.s.core.session.SessionRegistry` to maintain a list of active HTTP sessions and the authenticated users with which they are associated. As sessions are created and expired, the registry is updated in real-time, based on the session lifecycle events published by the `HttpSessionEventPublisher` to track the number of active sessions per authenticated user.
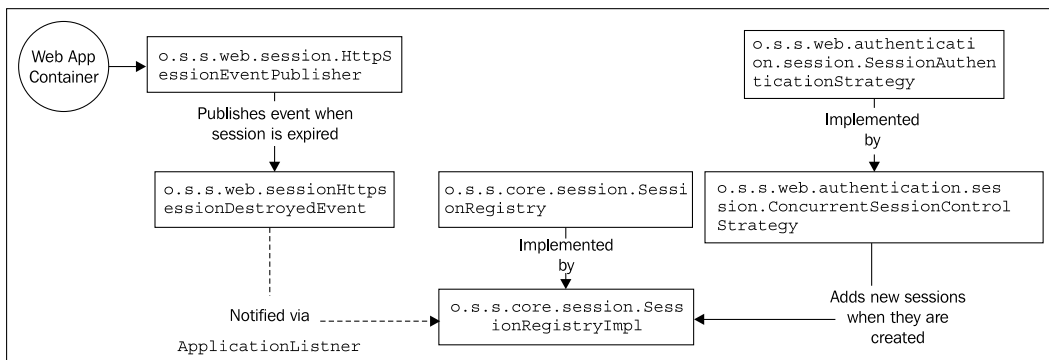
An extension of the `SessionAuthenticationStrategy`, `o.s.s.web.authentication.session.ConcurrentSessionControlStrategy` is the method by which new sessions are tracked and a method by which concurrency control is actually enforced. Each time a user accesses the secured site, the `SessionManagementFilter` is used to check the active session against the `SessionRegistry`. If the user's active session isn't in the list of active sessions tracked in the `SessionRegistry`, the least recently used session is immediately expired.

The secondary actor in the modified concurrent session control filter chain is the `o.s.s.web.session.ConcurrentSessionFilter`. This filter will recognize expired sessions (typically, sessions which have been expired either by the servlet container or forcibly by the `ConcurrentSessionControlStrategy`) and notify a user that their session has expired.

Now that we have understood how concurrent session control works, it should be easy for us to reproduce a scenario in which it is enforced.

Testing concurrent session control

As we did with verifying session fixation protection, we will need to access two web browsers. Follow these steps:

1. In Internet Explorer, log into the site as user guest.
2. Now, in Firefox, log into the site as the same user (guest).
3. Finally, go back to Internet Explorer, and take any action. You will see a message indicating that your session has expired.

The following message will appears:
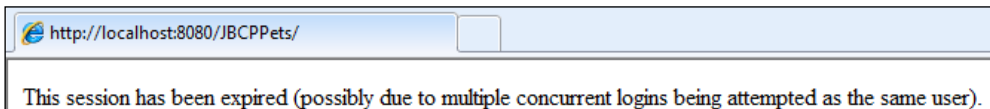
Not very friendly, but it illustrates that the session has been forcibly expired by the software. If you were an attacker, you'd be quite upset right now! If you were a legitimate user, however, you'd probably be confused, because this is obviously not a friendly method of being notified that JBCP Pets is keeping an eye on your security!

> Concurrent session control tends to be a very difficult concept for new Spring Security users to grasp. Many users try to implement it without truly understanding how it works, and what the benefits are. If you're trying to enable this powerful feature, and it doesn't seem to be working as you expect, make sure you have everything configured correctly, and then review the theoretical explanations in this section—hopefully they will help you understand what may be wrong!

In the event this occurs, we should probably redirect the user to the login page, and provide them a message to indicate what went wrong.

Configuring expired session redirect

Fortunately, there is a simple method of directing users to a friendly page (typically, the login page), when they are flagged by concurrent session control—simply specify the `expired-url` attribute, and set it to a valid page in your application:

```
<http auto-config="true" use-expressions="true">
<!-- ... -->
  <session-management>
    <concurrency-control max-sessions="1" expired-url=
    "/login.do?error=expired"/>
  </session-management>
</http>
```

In the case of our application, this will redirect the user to the standard login form, and we could alter the page to display a friendly message indicating that we determined they had multiple active sessions, and should log in again. Of course, the URL is completely arbitrary, and should be adjusted to match your application's needs!

Other benefits of concurrent session control

Another benefit of concurrent session control is that the `SessionRegistry` exists to track active (and, optionally, expired) sessions. This means that we can get runtime information about what user activity exists in our system (for authenticated users, at least).

You can even do this if you don't want to enable concurrent session control. Simply set `max-sessions` to the value of `-1`, and session tracking will remain enabled, even though no maximum will be enforced.

Let's look at a couple simple ways to use this.

Displaying a count of active users

You've probably seen online forums which display a count of currently active users in the system. With session registry tracking (through concurrent session control) enabled, this is trivial to display on each page of the application.

Let's add a simple method and bean auto-wiring to `BaseController`.

```
@Autowired
  SessionRegistry sessionRegistry;

@ModelAttribute("numUsers")
  public int getNumberOfUsers() {
    return sessionRegistry.getAllPrincipals().size();
  }
```

We can see that this exposes an attribute to be used in a Spring MVC JSP page, so we'll add a footer, `footer.jsp`, to the JBCP Pets site to take advantage of this.

```
<div id="footer">
  ${numUsers} user(s) are logged in!
</div>
</body>
</html>
```

Once you restart the application and log in, you can see that the active user count is updated at the bottom of every page.
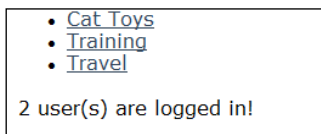
Trivial, but it illustrates the benefit of session tracking as part of the Spring Security framework—especially as you can take advantage of this functionality right out of the box!

We can do some more advanced data gathering from the `SessionRegistry`, which can be helpful for administrators—let's see this now.

Displaying information about all users

The `SessionRegistry` tracks information about all active user sessions. If we want to empower our site administrators, we can very easily put together a page listing all active users, and the last time they interacted with the site.

Let's add a new method to `AccountController` (although such a function would typically be added in an administrative area, we can suspend our belief that JBCP Pets is a real site for a moment!), which will review the information in the `SessionRegistry` and collect current session information.

```
@RequestMapping("/account/listActiveUsers.do")
public void listActiveUsers(Model model) {
  Map<Object,Date> lastActivityDates = new HashMap<Object, Date>();
  for(Object principal: sessionRegistry.getAllPrincipals()) {
  // a principal may have multiple active sessions
  for(SessionInformation session :
    sessionRegistry.getAllSessions(principal, false))
    {
    // no last activity stored
```

```
      if(lastActivityDates.get(principal) == null) {
        lastActivityDates.put(principal, session.getLastRequest());
      } else {
      // check to see if this session is newer than the last stored
        Date prevLastRequest = lastActivityDates.get(principal);
        if(session.getLastRequest().after(prevLastRequest)) {
          // update if so
          lastActivityDates.put(principal,
          session.getLastRequest());
        }
      }
    }
  }
  model.addAttribute("activeUsers", lastActivityDates);
}
```

This method utilizes two API methods on `SessionRegistry`.

- `getAllPrincipals`: Returns a `List` of `Principal` objects (typically `UserDetails` objects) with active sessions.

- `getAllSessions(principal, includeExpired)`: Returns a `List` of `SessionInformation` objects with information about each session for the given `Principal`. Can also include information about expired sessions.

Given this explanation of the `SessionRegistry` API methods, the logic of the `listActiveUsers` method is pretty straightforward—review all active users in the registry, and find the session with the most recent activity on it. The combination of `Principal` and last activity timestamp is inserted into a `Map` for display in the UI.

The UI page becomes very simple using JSTL constructs. Create `listActiveUsers.jsp` in the `WEB-INF/views/account` directory, with the following contents (we'll omit the header and footer for brevity):

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<h1>Active Users</h1>
<ul>
  <c:forEach items="${activeUsers}" var="uinfo">
    <li><strong>${uinfo.key.username}</strong>
    / Last Active: <strong>${uinfo.value}</strong></li>
  </c:forEach>
</ul>
```

What we end up with, when we navigate to `http://localhost:8080/JBCPPets/account/listActiveUsers.do`, is something like the following:

You can certainly see the power of the `SessionRegistry` for these types of trivial examples. It is even possible to extend the `SessionRegistry` to track additional information about user activity, such as last page viewed, last action taken, and so on—very helpful for administrative interfaces built on top of Spring Security!

Understanding and configuring exception handling

Spring Security uses a simple dispatcher pattern to translate exceptions thrown by the framework into concrete actions that affect the processing of a user's request to a secured resource. The `o.s.s.web.access.ExceptionTranslationFilter`, one of the last servlet filters in the standard Spring Security filter chain, is responsible for examining exceptions thrown during the authentication and authorization processes (in `FilterSecurityInterceptor`, the culmination of the filter chain), and reacting appropriately to them.

The standard `ExceptionTranslationFilter` provides dispatching for three general classes of failure, as illustrated in the following diagram:

Filter Chain

Servlet Filter

Servlet Filter

o.s.s.web.access.ExceptionTranslationFilter

Exception

Servlet Filter

Handle Exception

Is Authentication Exception?

Yes — Save Request for post-authentication (SavedRequest)

No

Is AccessDenied Exception?

No — Bubble Exception up filter chain

Yes

o.s.s.web.AuthenticationEntryPoint

Is User Anonymous?

Yes

No

Send user to login page or other authentication mechanism

Login mechanism depends on authentication type (form or other advanced type)

o.s.s.web.access.AccessDeniedHandler

Is `errorPage` Defined?

Yes — Redirect to errorPage

This refers to a name property of AccessDeniedHandler

No

Return HTTP 403 Response

We can see that `ExceptionTranslationFilter` handles the following cases:

- `AuthenticationException` is thrown, and the user is required to log in (in most cases—depending on the logic of the `AuthenticationEntryPoint`, as we'll read later in this chapter)

- `AccessDeniedException` is thrown, and the user hasn't logged in

- `AccessDeniedException` is thrown, and the user has logged in, in which case, the user is shown either an error page or a generic **HTTP 403** response

Let's dive into the configuration of `AccessDeniedHandler`.

Configuring "Access Denied" handling

To this point, when authenticated users have been denied access to a secured resource because they lack the `GrantedAuthority` or other required rights, they have been presented with the servlet container's default **HTTP 403** (**Access Denied**) page. This page is the result of actions taken by the default `o.s.s.web.access.AccessDeniedHandler`, which is invoked by the `ExceptionTranslationFilter` in response to an `AccessDeniedException` thrown by the framework.

While this plain error page is effective, it's not particularly attractive or user friendly. It would be better if we were able to tie this page into the overall look and feel of our site, and provide the user with some information about what happened.

A very basic method of handling the access denied report to the user is handled by configuring a custom URL to which Spring Security will forward the user whose request is denied. We note that this functions similarly to the user's initial login request being forwarded to the `login-page` declared on the `<form-login>` element.

Configuring an "Access Denied" destination URL

Configuring the URL to which a user will be forwarded is the easiest part of this exercise. Simply add a new element, `<access-denied-handler>`, within the `<http>` declaration, which declares our access denied handling URL as follows:

```
<http auto-config="true" ...>
  <access-denied-handler error-page="/accessDenied.do"/>
</http>
```

Hold tight—this won't work yet, because we haven't associated this URL with any code in our Spring MVC application. We'll need to augment the code in our `LoginLogoutController` to handle this additional URL, and to push some useful information into the model for our view to present to the user.

Adding controller handling of AccessDeniedException

We'll need to add a controller action handler method to respond to the URL that we just configured. Additionally, we'll inspect the `AccessDeniedException` and extract some details that may be of help to the user when they see our custom access denied page.

```
@Controller
public class LoginLogoutController extends BaseController{
  // Ch 6 Access Denied
```

```
@RequestMapping(method=RequestMethod.GET, value="/accessDenied.do").
public void accessDenied(ModelMap model,
HttpServletRequest request) {
  AccessDeniedException ex = (AccessDeniedException)
  request.getAttribute(AccessDeniedHandlerImpl
  .SPRING_SECURITY_ACCESS_DENIED_EXCEPTION_KEY);
  StringWriter sw = new StringWriter();
  model.addAttribute("errorDetails", ex.getMessage());
  ex.printStackTrace(new PrintWriter(sw));
  model.addAttribute("errorTrace", sw.toString());
}
}
```

Be aware that the reference to `AccessDeniedHandlerImpl` is required to retrieve the name of the `request` attribute that is used to temporarily store the exception thrown during the scope of the current request.

Unfortunately, the `AccessDeniedException` does not typically provide enough detail in its message to be extremely helpful to system administrators or the users themselves. You may be interested in examining the use of `AccessDeniedException` in the Spring Security framework and possibly extending it to provide more contextual information about what the user was trying to do when an authorization check was declined.

Writing the Access Denied page

With the controller in place, the **Access Denied** page is trivial to write.

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
  pageEncoding="ISO-8859-1"%>
<jsp:include page="common/header.jsp">
  <jsp:param name="pageTitle" value="Access Denied"/>
</jsp:include>

<h1>Access Denied</h1>
<p>
  Access to the specified resource has been denied for
  the following reason: <strong>${errorDetails}</strong>.
</p>
<em>Error Details (for Support Purposes only):</em><br />
<blockquote>
  <pre>${errorTrace}</pre>
</blockquote>
<jsp:include page="common/footer.jsp"/>
```

You can see that we have used the `errorDetails` and `errorTrace` model variables that we set in the controller. While not beautiful, this page serves its purpose to direct the user to other areas of the site, through the common site navigation, and also gives them some indication about what may have gone wrong.

What causes an AccessDeniedException

When planning for exception handling, it's important to do some analysis and understand the origin of a target exception. It's common for users of Spring Security to get confused between `AccessDeniedException` (which results in an **HTTP 403** page, by default) and `AuthenticationException`, which is typically thrown when the user hasn't been authenticated at all. The following guidelines may help indicate when each type of exception is thrown by the framework:

Exception type	Who throws it and why?		
`AuthenticationException`	•	An `AuthenticationProvider`, when supplied credentials are invalid, or the account is disabled or expired	
	•	A `DaoAuthenticationProvider`, when an error occurs accessing the DAO data store	
	•	`RememberMeServices`, when the remember me cookie presented has been tampered with	
	•	Various specialized authentication classes (CAS, NTLM, and so on.) for scenario-specific use cases	
`AccessDeniedException`	An `AccessDecisionManager`, when the configured `Voter` votes to deny access—note that this can happen in any voting scenario, including web URL access or method-level authorization checking.		

Keep in mind that the `ExceptionTranslationFilter` we mentioned prior to this exercise is the key bit that differentiates between the two types of exceptions as they relate to the flow of the user's request and the responses of the application to the user.

Pay attention to which filters precede the `ExceptionTranslationFilter` in the filter chain. The `ExceptionTranslationFilter` will be able to handle and react to only those exceptions that are thrown below it in the filter chain execution stack. Users often get confused, especially when adding custom filters in the incorrect order, as to why the expected behavior differs from their application's actual exception handling—in many of these cases, the order of the filters is to blame!

While the out of the box workflow is predictable and sufficient in most cases, you may find that you need to customize exception handling, especially if you introduce custom subclasses of the base exceptions, which require special treatment by the filter chain.

The importance of the AuthenticationEntryPoint

The `AuthenticationEntryPoint` (which we saw as a supporting bean in the workflow of the `ExceptionTranslationFilter`) plays a crucial role in the handling of unauthenticated user requests. When the `ExceptionTranslationFilter` determines that authentication is required, it reaches out to the `AuthenticationEntryPoint` to inquire what should happen next. In the case of form-based authentication, the `o.s.s.web.authentication.LoginUrlAuthenticationEntryPoint` is responsible for forwarding the user to the login form.

We will see in later chapters that the `AuthenticationEntryPoint` is used in a variety of authentication mechanisms for more specialized behavior—for example, in Central Authentication Service (CAS) single-sign on, the `AuthenticationEntryPoint` will ensure the user is forwarded to the CAS portal for authentication.

In many circumstances, when implementing custom integrations with Spring Security and a third-party authentication system (external to the web application), you will need to implement an `AuthenticationEntryPoint` yourself.

Configuring Spring Security infrastructure beans manually

If you're working in a very complex environment where the baseline Spring Security functionality—although very robust—doesn't cover everything, you may end up at a point where you'll need to build the entire Spring Security filter chain and supporting infrastructure from the ground up yourself. This is an area that is partially documented in the Spring Security reference documentation, but tends to trip up a lot of people. Some refer to this type of configuration as the "alternate universe" of Spring Security.

Building and wiring all the requisite beans yourself, provides you with a very high degree of flexibility and customization that the `security` namespace `<http>`-style configuration won't allow.

> We aren't kidding when we say that building up the requisite beans is complex. Even a bare-bones implementation can require upwards of 25 individual bean definitions, with a clear understanding on the part of the developer about which beans are dependent on others, and what all the properties on each bean are for.
>
> Keep in mind that once you step off the cliff of the convenient security XML namespace-based configuration, you are less insulated from changes in the Spring Security codebase and overall dependency architecture. The `security` XML namespace provides a welcome layer of abstraction and will serve most needs perfectly well.

Hopefully at this point in the book, you have understood both the architecture of request processing and the major components involved in it; hence, you won't be shocked at the large number of beans needing configuration. Purely to get a better understanding of all the components that are under the surface, it can be a useful exercise to work through configuring each and every bean, to see the final, working configuration.

We'll spend some time here illustrating the major pieces required when setting up the Spring Security infrastructure yourself. Note that in some cases, we may reduce some of the detail of less interesting beans, where it doesn't really need to be explained; however, the full configuration file required to support this is included with the chapter source code as `dogstore-explicit-base.xml`. Let's walk through the major configuration steps now.

A high level overview of Spring Security bean dependencies

We don't want to jump right into configuring beans without giving you a very high level view of the major beans involved in the manual setup of the Spring Security beans. Here's a dependency diagram indicating the beans we'll be configuring, and how they interact with each other (the diagram is included with the source code of this chapter in full size, for your reference):
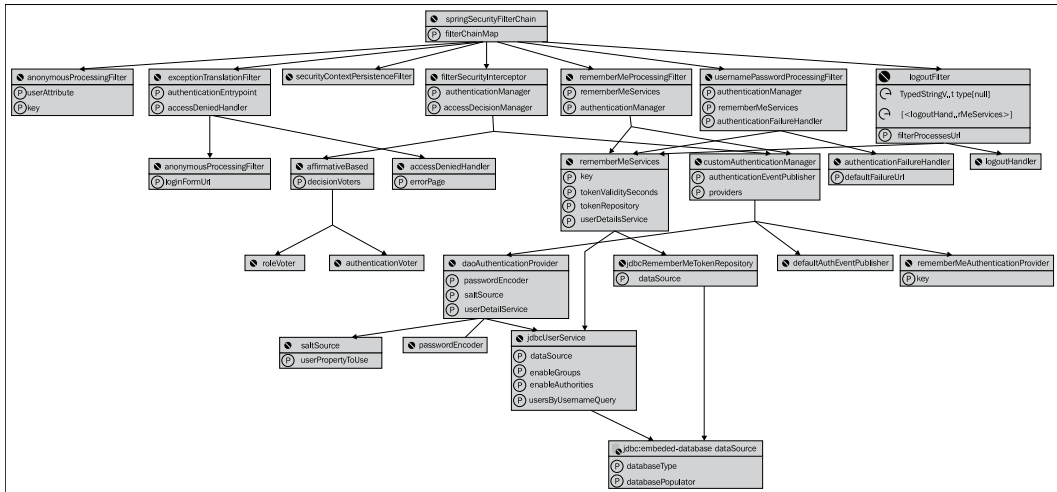
Keep in mind that this diagram covers much more than the bare minimum required to get a site up and running. We'll incrementally illustrate how to add all of these beans, starting from the minimum set, and working our way to a full stack with functionality comparable to what we've configured using the security namespace.

Reconfiguring the web application

For the sake of clarity, we'll create a brand new XML configuration file for this style of configuration. This will allow us to be very precise as to what exactly is required, and remove some of the bits we've commented and uncommented as we've worked through exercises in previous chapters.

To do this, we'll need to reconfigure our Spring `ApplicationContext` to point to this new file. We'll call the file `dogstore-explicit-base.xml`, and update our `web.xml` file to reference it as follows:

```xml
<web-app ...>
  <display-name>Dog Store</display-name>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
      /WEB-INF/dogstore-explicit-base.xml
    </param-value>
  </context-param>
```

We'll no longer need the separate `dogstore-security.xml` file, which we had created for convenience when using XML declarations in the `security` namespace. The majority of our configuration will be using standard Spring bean wiring syntax, with a sprinkling of `security` namespace decorators thrown in.

Configuring a minimal Spring Security environment

We'll start with the bare minimum required to get the application up and running again—this means no remember me, no logout, and no exception handling. This allows us to focus on the bare minimum required to get Spring Security started.

First, we'll have to declare the servlet filter chain that Spring Security will use when intercepting requests as follows:

```xml
<bean id="springSecurityFilterChain"
  class="org.springframework.security.web.FilterChainProxy">
  <security:filter-chain-map path-type="ant">
    <security:filter-chain pattern="/**" filters="
      securityContextPersistenceFilter,
      usernamePasswordAuthenticationFilter,
      anonymousAuthenticationFilter,
      filterSecurityInterceptor" />
  </security:filter-chain-map>
</bean>
```

You can see that we're already referencing the security namespace here. Though it's possible to manually configure the required bean properties to set up path pattern matching and filter list combinations, the syntax for doing so using the security namespace `filter-chain-map` decorator is more concise and convenient. If we map this to the `<http>` style of configuration, we will note the following additional configurable elements:

- The setup of the default filter chain is implicit in the overall processing of the `<http>` element and isn't directly configurable. Although the use of the `security` namespace's `<custom-filter>` declaration allows a very high degree of flexibility when overriding or augmenting the standard filter chain, it doesn't let you get at the `FilterChainProxy` itself.

- The ability to alter the filter chain based on URL pattern isn't available with the `<http>` style declaration. This can be helpful if certain portions of your application don't require certain types of processing, and you want to make the overhead of multiple filter invocations as small as possible.

It's important to note that, unlike some configuration elements in Spring (notably, `contextConfigLocation` in `web.xml`), the use of commas between filter names is required here.

> The order of filters is important—as discussed in Chapter 2, certain filters must come before others. Unless you have special, specific needs, please refer to the table in Chapter 2, when adding standard filters into a manually configured filter chain, to ensure that you include them at the appropriate positions. Failure to include filters in the correct order may cause unexpected or undesired application behavior that can be very difficult to debug!

You'll notice that the `<filter-chain>` element refers to many logical bean definitions that we've not yet defined. Let's define them now, and explain each one in detail.

Configuring a minimal servlet filter set

There are two sets of objects we'll need to configure in order to support the filter chain described earlier.

First, the servlet filters themselves must be configured. These define the processing of user requests as they enter the web application—the difference from our use of the security namespace is that we are closer to the metal here, and explicitly defining much about the filters that was hidden with the simpler configuration mechanism we'd used until now.

Secondly, the servlet filters rely on a supporting set of security infrastructure beans. Many of these classes will be familiar to you, as we've covered their functionality in the architecture and functional overview in chapters 1 through 5, but the mechanism of configuring these beans is new.

We'll handle the required filters first.

SecurityContextPersistenceFilter

The `SecurityContextPersistenceFilter` is used to set up the `SecurityContext`, which is used throughout the scope of a request to track the authentication information related to the requestor for the duration of the entire request. You may remember that we accessed the `SecurityContext` object in the last chapter when we wanted to get information about the currently authenticated `Principal` in our Java Spring MVC controller code.

Basic configuration of the `SecurityContextPersistenceFilter` with reasonable defaults for web session management requires the following configuration:

```
<bean id="securityContextPersistenceFilter"
      class="org.springframework.security.web.context
      .SecurityContextPersistenceFilter/>
```

There's a lot going on behind the scenes here, and we can get quite sophisticated with configuration, as to how the HTTP session is managed. For now, we'll accept this filter with defaults and move on, but we'll examine session-related configuration options later in this chapter.

UsernamePasswordAuthenticationFilter

As we discussed in detail in Chapter 2, `UsernamePasswordAuthenticationFilter` is used to process form submission and check with the authentication store for valid credentials. Explicit configuration of this filter, mirroring the security namespace configuration, is as follows:

```
<bean id="UsernamePasswordAuthenticationFilter"
  class="org.springframework.security.web
  .authentication.UsernamePasswordAuthenticationFilter">
  <property name="authenticationManager"
  ref="customAuthenticationManager"/>
</bean>
```

You'll notice a pattern if you have the time to delve into the classes—there's much more that can be configured here. Again, we'll visit this once we get the basic configuration working. We have a forward reference to a bean called `customAuthenticationManager`—this is indeed the same `AuthenticationManager` that's normally auto-configured by the `<authentication-manager>` element in the security namespace. We'll need to configure this bean shortly.

AnonymousAuthenticationFilter

Our site needs to allow anonymous access. Although there are very specific conditions where the `AnonymousAuthenticationFilter` isn't required, typically it should be used, as it adds very little overhead to request processing. You may not recognize this filter, except for a brief mention in Chapter 2. This is because the configuration of the `AnonymousAuthenticationFilter` is largely obscured by the security namespace configuration.

The minimal configuration of the filter is as follows:

```
<bean id="anonymousAuthenticationFilter"
  class="org.springframework.security.web
  .authentication.AnonymousAuthenticationFilter">
  <property name="userAttribute"
  value="anonymousUser,ROLE_ANONYMOUS"/>
  <property name="key" value="BF93JFJ091N00Q7HF"/>
</bean>
```

Both of the properties listed here are required. The `userAttribute` property indicates the username and `GrantedAuthority` provided to the anonymous user. The username and `GrantedAuthority` may be used in our application to validate whether or not the user is anonymous. The key may be arbitrarily generated, but needs to be applied in a bean (the `o.s.s.authentication.AnonymousAuthenticationProvider`) that we'll configure later.

FilterSecurityInterceptor

The final filter in our basic processing chain is the filter that is ultimately responsible for checking the `Authentication` object, which is the result of prior processing by the configured security filters. This is the filter that determines whether or not a particular request is ultimately rejected or accepted.

Let's review the configuration for this filter and try working through the exercise of mapping this filter to its corresponding location in the security namespace.

```
<bean id="filterSecurityInterceptor"
  class="org.springframework.security.web.access
  .intercept.FilterSecurityInterceptor">
  <property name="authenticationManager"
  ref="customAuthenticationManager"/>
  <property name="accessDecisionManager" ref="affirmativeBased"/>
  <property name="securityMetadataSource">
    <security:filter-security-metadata-source>
      <security:intercept-url pattern="/
        login.do" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
      <security:intercept-url pattern="/
```

```
        home.do" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
      <security:intercept-url pattern="/
        account/*.do" access="ROLE_USER"/>
      <security:intercept-url pattern="/*" access="ROLE_USER"/>
    </security:filter-security-metadata-source>
  </property>
</bean>
```

Think for a moment before reading on. That's right, these look like the same `<intercept-url>` declarations we used in the security namespace's `<http>` configuration. The pattern matching and format of the access declarations is exactly the same, but you'll note in this case that we're using some configuration magic to use these same declarations in the context of setting a normal Spring bean property.

The `<filter-security-metadata-source>` element is responsible for configuring the `SecurityMetadataSource` implementation used by the `FilterSecurityInterceptor`, including the URL declarations and roles required to access them.

> **Effective use of XML namespaces**
>
> It's common for users who are not familiar with advanced configuration of Spring to be confused at this point. Spring XML configuration makes effective use of XML namespaces to provide clear component-based ownership of different elements in the configuration file. An element indicates that it is in a namespace when it is prefixed with a colon (`:`), followed by the element name. So, if we look at `<security: intercept-url>`, we can see that this element's name is `intercept-url`, and it is in the XML namespace called `security`. XML namespaces are generally declared at the top of an XML file, with an arbitrary prefix assignment, associated with a URI. For example, the security namespace may be declared as `xmlns:security="http://www. springframework.org/schema/security"`.
>
> What about elements with no declared namespace? These are assigned to the default namespace for the XML document. The default namespace is indicated by the `xmlns` attribute—in the case of the `dogstore-explicit-base.xml` file, the default namespace is declared as `xmlns="http://www.springframework.org/schema/beans"`.
>
> Elements with no namespace prefix will be implicitly assigned to the default namespace. Understanding how XML namespaces really work will save you hours of headaches when building complex Spring and Spring Security configuration files.

We've now configured all the filters required for a minimal, explicitly configured filter chain. These filters can't exist by themselves—there are a few supporting objects that are required as well.

Configuring a minimal supporting object set

The supporting objects that are required to round out our minimal configuration are Spring beans that we've already (save one) configured in exercises in prior chapters—thus we'll save time explaining their purpose here.

The following is a set of bean definitions needed to complete the minimal supporting object set and start up the application:

```
<bean class="org.springframework.security.access
   .vote.AffirmativeBased" id="affirmativeBased">
  <property name="decisionVoters">
    <list>
      <ref bean="roleVoter"/>
      <ref bean="authenticatedVoter"/>
    </list>
  </property>
</bean>
<bean class="org.springframework.security.access
   .vote.RoleVoter" id="roleVoter"/>
<bean class="org.springframework.security.access
   .vote.AuthenticatedVoter" id="authenticatedVoter"/>
<bean id="daoAuthenticationProvider" class="org.springframework
   .security.authentication.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
<bean id="anonymousAuthenticationProvider" class="org.springframework.
security.authentication.AnonymousAuthenticationProvider">
    <property name="key" value="BF93JFJ091N00Q7HF"/>
</bean>
```

This configuration provides us with a minimal configuration that supports anonymous browsing, login, and database-backed authentication (note that we omitted the required `jdbcUserService` and `dataSource` beans for brevity—the definitions of those beans haven't changed from their prior definitions). Remember that the `key` property of the `AnonymousAuthenticationProvider` must match the `key` property that we previously defined for the `AnonymousAuthenticationFilter`.

The one bean that's required, which we hadn't configured before (because the security namespace configuration doesn't allow it), is the `AuthenticationManager`. We can define this bean as follows:

```
<bean id="customAuthenticationManager" class="org.springframework
   .security.authentication.ProviderManager">
  <property name="providers">
    <list>
```

```
            <ref local="daoAuthenticationProvider"/>
            <ref local="anonymousAuthenticationProvider"/>
        </list>
    </property>
</bean>
```

Although the configuration of the `AuthenticationManager` may look trivial here, explicit configuration of this bean is the key to many useful enhancements and opportunities for extension of the framework, which we'll cover over the remainder of this chapter.

After doing all this work to configure beans by hand, our site still doesn't support some of the functionality we had when using the security XML namespace, including logout functionality, friendly exception handling, password salting, or remember me. So, why is this valuable?

Advanced Spring Security bean-based configuration

As we've hinted at on several previous pages, bean-based configuration of Spring Security, although complex, provides a level of flexibility that will be required for complex applications with requirements above and beyond what the security XML namespace-style configuration can allow.

We'll spend this section elucidating some of the configuration options available, and how they might be useful. Although we cannot provide details on every possible property, we encourage you to start exploring, based on what you've learned in this chapter and prior chapters, and consult the Javadoc and Spring Security source code. You are limited only by your knowledge of how the framework is put together, and your imagination!

Adjusting factors related to session lifecycle

Spring Security has a variety of bits that can dramatically affect the lifecycle of your users' `HttpSession`. Many of these are available only when you have configured the relevant classes as Spring beans. The following table lists the bean attributes that can affect session creation and destruction:

Class	Attribute	Default	Description
Abstract Authentication ProcessingFilter (parent of UsernamePassword Authentication Filter)	allowSessionCreation	true	If true, a new session may be created on a failed authentication attempt (to store the exception).
UsernamePassword AuthenticationFilter	allowSessionCreation	true	If true, this particular filter may create a session to store the last username attempted.
SecurityContext LogoutHandler	invalidateHttpSession	true	If true, the `HttpSession` will be invalidated (refer to the Servlet spec for details on session invalidation).
SecurityContext PersistenceFilter	forceEager SessionCreation	false	If true, the filter will create a session before executing the remainder of the filters in the chain.
HttpSessionSecurity ContextRepository	allowSessionCreation	true	If true, the `SecurityContext` can be saved to the session if one doesn't exist by the time the user's request ends.

Depending on the needs of your application, it may be prudent to analyze the size of a typical user's session— both authenticated and unauthenticated—and adjust the expected session lifecycle accordingly.

Manual configuration of other common services

There are still a few more services, which we've grown used to taking advantage of, with the security namespace automatic configuration. Let's work through adding these so that we have a completely configured baseline implementation to work in.

Declaring remaining missing filters

We'll augment the manually-configured filter chain with the three services that we haven't yet configured. These include filters to handle logout, remember me, and exception translation. Once we've completed these filters, we'll have a full stack and can delve into some interesting configuration options available with this setup.

The following missing filters are added to our filter chain:

```xml
<bean id="springSecurityFilterChain"
  class="org.springframework.security.web.FilterChainProxy">
  <security:filter-chain-map path-type="ant">
    <security:filter-chain pattern="/**" filters="
    securityContextPersistenceFilter,
    logoutFilter,
    usernamePasswordAuthenticationFilter,
    rememberMeAuthenticationFilter,
    anonymousAuthenticationFilter,
    exceptionTranslationFilter,
    filterSecurityInterceptor" />
  </security:filter-chain-map>
</bean>
```

As we have done before, we now have to declare these Spring beans and configure them, along with any dependent service beans.

LogoutFilter

The basic declaration of the `LogoutFilter` (used to respond to events on the virtual URL `/j_spring_security_logout`, by default) is as follows:

```xml
<bean id="logoutFilter" class="org.springframework.security
  .web.authentication.logout.LogoutFilter">
  <!-- the post-logout destination -->
  <constructor-arg value="/"/>
    <constructor-arg>
      <array>
        <ref local="logoutHandler"/>
      </array>
    </constructor-arg>
  <property name="filterProcessesUrl" value="/logout"/>
</bean>
```

You'll note that the method of construction for the `LogoutFilter` is different from any of the other authentication filters, utilizing constructor arguments in addition to bean properties. The first constructor argument is the URL to which the user will be sent after logout, and the second argument is a reference to a `LogoutHandler` instance.

The philosophy of constructors in dependency injection

If you have worked in the Spring world for a while, you have probably discussed the proper object-oriented approach for managing required dependencies. Although setters are convenient for users of Spring, object-oriented purists often argue that if a particular dependency is required for a class to function, it should be part of a constructor signature (think of how you used to write classes prior to Spring).

As Spring Security has evolved over the years, different authors have had different opinions about this, and we occasionally run into one or two inconsistencies of this nature. Both styles of dependency modeling work - which style you use really depends upon what type of modeling for required dependencies you prefer.

We'll declare a simple `LogoutHandler` as follows:

```
<bean id="logoutHandler" class="org.springframework.security
    .web.authentication.logout.SecurityContextLogoutHandler"/>
```

You may recognize this `LogoutHandler` from the table regarding session handling we saw a few pages ago. One benefit of explicitly configuring the logout handler is that you can choose to adjust the default session handling behavior on logout. Once we've configured these two bits, the **Log Out** link should begin to work again.

Remember that with our security namespace configuration in Chapter 3, we had revised our application to use a logout URL which wouldn't so obviously tie our application to Spring Security. That's why we're overriding the `filterProcessesUrl` property in this bean definition to keep the configuration underneath our application consistent, as we change the method of configuration from `security` namespace declared to explicit bean declarations!

RememberMeAuthenticationFilter

You'll recall enabling remember me functionality in Chapter 3 as well. We'd like to retain the helpful remember me feature, even in our bean-based configuration. This involves a combination of a new filter, new supporting beans, and hooks into several other beans. First, let's have a look at the filter.

```
<bean id="rememberMeAuthenticationFilter"
  class="org.springframework.security.web
  .authentication.rememberme.RememberMeAuthenticationFilter">
  <property name="rememberMeServices" ref="rememberMeServices"/>
  <property name="authenticationManager"
  ref="customAuthenticationManager" />
</bean>
```

You'll note the reference to `rememberMeServices`, which hasn't been defined yet. Let's define this bean now:

```
<bean id="rememberMeServices"
  class="org.springframework.security.web.authentication
  .rememberme.PersistentTokenBasedRememberMeServices">
  <property name="key" value="jbcpPetStore"/>
  <property name="tokenValiditySeconds" value="3600"/>
  <property name="tokenRepository"
  ref="jdbcRememberMeTokenRepository"/>
  <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

Many of the configuration properties on the `RememberMeServices` implementation have very similar names in the security namespace style of configuration. The major difference between these two methods is that the details of the `RememberMeServices` are abstracted from the configured application details, while in the bean-based configuration, the bean declarer must be very aware of all the beans involved in the remember me feature. In case you're confused, refer back to chapters 3 and 4 for details on the classes and flow involved in the remember me feature.

We've got one more bean reference to wire up to the remember me token repository. This bean definition is simple, as we see here:

```
<bean id="jdbcRememberMeTokenRepository"
  class="org.springframework.security.web
  .authentication.rememberme.JdbcTokenRepositoryImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

Finally, we'll need to declare the `AuthenticationProvider`, which can handle the remember me authentication requests. This declaration looks as you might expect:

```
<bean id="rememberMeAuthenticationProvider"
  class="org.springframework.security
  .authentication.RememberMeAuthenticationProvider">
  <property name="key" value="jbcpPetStore"/>
</bean>
```

You may remember that the `key` attribute is shared between the `AuthenticationProvider`, for token validation, and the `RememberMeServices`, for token generation. Make sure these are set to the same value! You may want to set this property value from a properties file, using the Spring `PropertyPlaceholderConfigurer` utility class (or something similar).

We'll now need to wire this `AuthenticationProvider` into the list provided to our
`AuthenticationManager`.

```
<bean id="customAuthenticationManager"
  class="org.springframework.security
  .authentication.ProviderManager">
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
      <ref local="anonymousAuthenticationProvider"/>
      <ref local="rememberMeAuthenticationProvider"/>
    </list>
  </property>
</bean>
```

`RememberMeServices` must also be wired into the
`UsernamePasswordAuthenticationFilter`, so that the `RememberMeServices` can
handle explicit login success (remember me cookie is updated) or failure (remember
me cookie is removed).

```
<bean id="usernamePasswordAuthenticationFilter"
  class="org.springframework.security.web
  .authentication.UsernamePasswordAuthenticationFilter">
  <property name="authenticationManager"
  ref="customAuthenticationManager"/>
  <property name="rememberMeServices" ref="rememberMeServices"/>
</bean>
```

One last thing to remember (often forgotten by users configuring bean declarations
for Spring Security) is that the `RememberMeServices` also functions as a
`LogoutHandler`, which clears the user's cookie upon logout.

```
<bean id="logoutFilter"
  class="org.springframework.security.web
  .authentication.logout.LogoutFilter">
  <constructor-arg value="/"/>
  <constructor-arg>
    <array>
      <ref local="logoutHandler"/>
      <ref local="rememberMeServices"/>
    </array>
  </constructor-arg>
  <property name="filterProcessesUrl" value="/logout"/>
</bean>
```

With all these configurations in place, we've completed our work wiring the
remember me functionality explicitly. You probably weren't aware of how
much work the `<remember-me>` declaration was doing behind the scenes!

ExceptionTranslationFilter

The final servlet filter in the standard Spring Security filter chain is the `ExceptionTranslationFilter`. We recall the importance of this filter in handling overall flow of authentication and authorization in the application, from our discussion earlier in this chapter. This final filter requires one filter definition and two supporting beans.

```
<bean id="exceptionTranslationFilter"
  class="org.springframework.security.web
  .access.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint"
  ref="authenticationEntryPoint"/>
  <property name="accessDeniedHandler" ref="accessDeniedHandler"/>
</bean>
```

The supporting beans are declared as follows:

```
<bean id="authenticationEntryPoint"
  class="org.springframework.security.web
  .authentication.LoginUrlAuthenticationEntryPoint">
  <property name="loginFormUrl" value="/login.do"/>
</bean>
<bean id="accessDeniedHandler"
  class="org.springframework.security.web
  .access.AccessDeniedHandlerImpl">
  <property name="errorPage" value="/accessDenied.do"/>
</bean>
```

You will recognize the **Access Denied** page defined in the `errorPage` directive from the access denied configuration exercise earlier in this chapter. Similarly, the `loginFormUrl` maps to the `login-page` attribute in the `security` namespace we've seen earlier.

Explicit configuration of the SpEL expression evaluator and Voter

Let's look at the equivalent Spring bean configuration to the `use-expressions="true"` attribute of the `security` namespace:

```
<bean class="org.springframework.security
  .web.access.expression.DefaultWebSecurityExpressionHandler"
  id="expressionHandler"/>
```

Now, we need to set up the `Voter` to point to the expression handler as follows:

```
<bean class="org.springframework.security.web.access
  .expression.WebExpressionVoter" id="expressionVoter">
  <property name="expressionHandler" ref="expressionHandler"/>
</bean>
```

Finally, we'll need to point our `AccessDecisionManager` bean to use the `Voter`.

```
<bean class="org.springframework.security.access
  .vote.AffirmativeBased" id="affirmativeBased">
  <property name="decisionVoters">
    <list>
      <ref bean="expressionVoter"/>
    </list>
  </property>
</bean>
```

After we've completed all this configuration, we're right back where we started when we used the simple `use-expressions="true"` default settings! However, now that we've made the configuration explicit, we could substitute the default implementations of any or all of the classes involved with custom ones. We'll see an example of this later in the chapter.

Bean-based configuration of method security

In Chapter 5, we configured method security using the `<global-method-security>` declaration in the `security` namespace style of configuration. In moving to our explicit, bean-based configuration, we must configure the required primary and supporting classes to replicate the functionality of our `<global-method-security>` declaration.

We have developed the full bean configuration to enable method security, supporting all types of the annotations discussed in Chapter 5. As this configuration is relatively straightforward, with few interesting configurable properties, we have placed the entire configuration in *Appendix*, *Additional Reference Material* along with the `dogstore-explicit-base.xml` configuration file for this chapter!

Wrapping up explicit configuration

At this point, we've finished configuring the bean-based Spring Security stack and the application should be fully functional. If you'd like to experiment with the features of the security namespace-style configuration versus the bean configuration, it's easiest to adjust the Spring configuration file references in `web.xml` to point to one set of configuration files or the other.

- `security` namespace configuration is handled by `dogstore-base.xml` and `dogstore-security.xml` references in `web.xml`

- Bean-based configuration is handled by `dogstore-explicit-base.xml` reference in `web.xml`

From this point onwards in the book, we'll assume you're using the `security` namespace configuration, in following along with exercises. If certain features are available using bean-based configuration only, we'll note them explicitly. We'll also include some details about bean-based configuration, as we know that some developers prefer to have this level of control for the security framework.

Which type of configuration should I choose?

Hopefully, your head isn't reeling from all the configuration we just had to do. You may be wondering, for a typical project, which style of configuration you should choose. The answer, as you might expect, depends on the complexity of your project, and the degree to which you're looking to override or customize default elements of the Spring Security stack.

Configuration type	Benefits
`security` namespace	• Powerful, abbreviated syntax for common web-based and method-based security configuration
	• Users need not know the details of how components interact behind the scenes to configure complex features
	• `security` namespace handler code detects and warns of many types of potential configuration issues
	• Significantly reduced chance of a missed configuration step

Configuration type	Benefits
Explicit Bean Definition	• Allows for ultimate flexibility in extension, override, and intentional omission of standard Spring stack
	• Allows for customized filter chains and authentication methods depending on URL patterns (using the `pattern` attribute of the `<filter-chain>` element). This may be required for scenarios with mixed web service or REST authentication and user-based authentication.
	• Does not directly tie configuration files to Spring Security namespace handling.
	• Authentication Manager may be explicitly configured or overridden.
	• Many more configuration options exposed than with the simpler security namespace.

With most projects, it makes sense to start with the `security` namespace configuration, and continue to use it if possible, until or unless there is a capability your application requires which isn't supported. Keep in mind that there is a significantly added complexity when configuring all required beans and inter-bean dependencies yourself, and before you start, you should have a very clear understanding of the Spring Security architecture (and probably underlying code as well!).

Authentication event handling

One significant feature that is available only with bean-based configuration is custom processing of authentication events. Authentication events are implemented using the Spring event publishing mechanism, which is based on the `o.s.context.ApplicationEvent` event model. The Spring event model isn't widely used, but can be very helpful—especially in authenticated systems—when you want to tie specific behavior to actions within the authentication realm.

The event is a typical publish-subscribe model, where notification of subscribers is handled by the Spring runtime itself. It's important to note that by default, the Spring event model is synchronous, so the runtime overhead of any subscribed listeners directly impacts the perceived performance of the request that generated the event.

At `ApplicationContext` initialization time, Spring will introspect all configured beans for the presence of the `o.s.context.ApplicationListener` interface. References to these beans are then retained by the configured `o.s.context.event.ApplicationEventMulticaster` that manages publication of events during runtime as they are published by the `o.s.context.ApplicationEventPublisher`. This infrastructure has been around for quite some time (since Spring 1.1), so there's lots of documentation available, should you wish to explore this area of Spring further.

The following diagram illustrates how the event publishing process comes together:
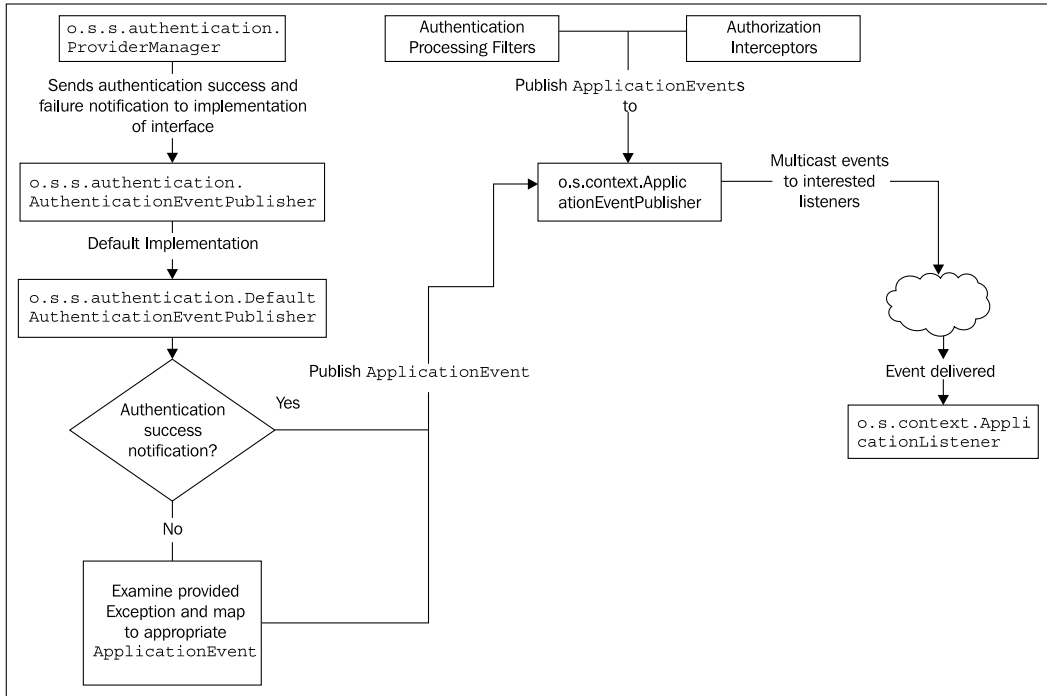
While there isn't wide use of authentication events out of the box in Spring Security (in fact, the only notable use is concurrent session tracking, as we discussed earlier in this chapter), tying custom listeners to authentication events can be a very convenient way to implement auditing, management alerts, or even sophisticated user activity tracking.

Let's work through the process of configuring a simple security event listener.

Configuring an authentication event listener

Using the shorthand `security` namespace configuration, you cannot configure an authentication event listener—it must be done with a Spring bean-based configuration because the `ApplicationEventPublisher` implementation is not enabled by default, and must be wired into the `AuthenticationManager`.

Declaring required bean dependencies

We'll first declare the `ApplicationEventPublisher` implementation as follows:

```
<bean id="defaultAuthEventPublisher"
  class="org.springframework.security.authentication
  .DefaultAuthenticationEventPublisher"/>
```

Next, we'll wire this into the `AuthenticationManager` implementation we're using.

```
<bean id="customAuthenticationManager"
  class="org.springframework.security
  .authentication.ProviderManager">
  <property name="authenticationEventPublisher" ref="defaultAuthEvent
Publisher"/>
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider"/>
      <ref local="rememberMeAuthenticationProvider"/>
    </list>
  </property>
</bean>
```

This is the entire configuration required. If you restart the application at this point, you will see nothing! That's because we haven't created a bean to listen to the events that are published. We'll do this now.

Building a custom application event listener

The `ApplicationListener` implementation is very straightforward, and uses the powerful Java generics infrastructure added to Spring 3 to support type safety. Our `ApplicationListener` will simply log the event received to standard output, but we'll explore a more interesting example in a later exercise.

The custom `ApplicationListener` is as follows:

```
package com.packtpub.springsecurity.security;
// imports omitted
  @Component
  public class CustomAuthenticationEventListener implements
```

```
        ApplicationListener<AbstractAuthenticationEvent> {
    @Override
    public void onApplicationEvent(AbstractAuthenticationEvent event) {
        System.out.println("Received event of type:
          "+event.getClass().getName()+": "+event.toString());
    }
}
```

You'll note that we used the shorthand `@Component` annotation here, but we could just as easily have explicitly defined the Spring bean in the XML configuration file.

Remember that `ApplicationListener` implementations must indicate what type of event they're interested in, and they indicate this in Spring 3 by declaring through the generic specified in the `ApplicationListener` interface reference. The Spring `ApplicationEventMulticaster` uses some smarts to introspect the class's interface implementation declaration and ensure the right events go to the right classes.

> **Brilliant generics**
>
> Those of you who are intrigued by the intricacy of generics handling and runtime introspection of generics should definitely explore the brilliant code behind the dispatching of `ApplicationEvent` using Spring's `o.s.context.event.GenericApplicationListenerAdapter`. Have a look and learn some new tricks with Java reflection!

Try starting up the application now and performing some typical activities such as log in, log out, and login failure. You can see that, as all these activities are performed, appropriate events will be fired and printed to the console.

Although we declared our `ApplicationListener` to receive all authentication events, in most scenarios this won't be practical, as depending on the level of use your system gets, there may be thousands of events fired per hour. By altering the generic qualifier in the `implements` clause on the class, we can narrow down our implementation to listen for a single type of event.

Out of the box ApplicationListeners

Spring Security ships with a couple of `ApplicationListener` implementations, which simply tie into the Apache Commons Logging loggers used by the rest of the Spring Security code. There are two `ApplicationListener` implementations, one for authentication events and one for authorization events. You can configure them as indicated in the following code:

```
<bean id="authenticationListener"
  class="org.springframework.security
  .authentication.event.LoggerListener"/>
<bean id="authorizationListener"
  class="org.springframework.security
  .access.event.LoggerListener"/>
```

These two listeners will output to Commons Logging loggers named after each respective class. A sample logged `AbstractAuthenticationFailureEvent` is similar to the following:

```
WARN - Authentication event
AuthenticationFailureBadCredentialsEvent: adb; details: org.
springframework.security.web.authentication.WebAuthenticationDetails@2
55f8: RemoteIpAddress: 127.0.0.1; SessionId: B20510F25464B109CE3AE94D9
FBF981E; exception: Bad credentials
```

These classes can be helpful as templates if you're writing your own, similar `ApplicationListener` to log interesting events!

Multitudes of application events

Spring Security provides a multitude of events intended to alert interested parties of all points in the lifecycle of a user authentication request. The available types of events, for which your application can listen, range from very broad (all authentication failures) to very narrow (has a user successfully authenticated by presenting a full set of credentials?). The full list of available events is documented in *Appendix*, *Additional Reference Material*. Some additional notes about exception handling and event listening are as follows:

- The mapping of authorization events to exceptions thrown by the framework is configurable with the `exceptionMappings` property of the `DefaultAuthenticationEventPublisher`

- Remember, as we saw earlier in this chapter, that tracking of `HttpSession` lifecycle is handled through a `web.xml` configuration change, and not directly by Spring

You can see that Spring Security's exception and event handling is quite robust, and allows for a number of interesting scenarios around tracking and responding to an activity, in your secured system.

Building a custom implementation of an SpEL expression handler

We'll illustrate a simple example of extending the base SpEL expression handler to provide an expression that will grant access if the current minute of the day is an even number. Although this is a contrived example, it illustrates all the steps required to implement your own custom SpEL expression methods.

Let's create a `com.packtpub.springsecurity.security.` `CustomWebSecurityExpressionRoot` class to set up a custom extension to `WebSecurityExpressionRoot`.

```
public class CustomWebSecurityExpressionRoot
   extends WebSecurityExpressionRoot {
   public CustomWebSecurityExpressionRoot
     (Authentication a, FilterInvocation fi) {
     super(a, fi);
   }

   public boolean isEvenMinute() {
     return (Calendar.getInstance().get(Calendar.MINUTE) % 2) == 0;
   }
}
```

Next, we'll need to build a class to implement `WebSecurityExpressionHandler`. We'll extend the `DefaultWebSecurityExpressionHandler` and override a single method to set up our `CustomWebSecurityExpressionRoot` in our `com.packtpub.` `springsecurity.security.CustomWebSecurityExpressionHandler` class.

```
public class CustomWebSecurityExpressionHandler
   extends DefaultWebSecurityExpressionHandler {
   public EvaluationContext createEvaluationContext
     (Authentication authentication, FilterInvocation fi) {
     StandardEvaluationContext ctx = (StandardEvaluationContext)
       super.createEvaluationContext(authentication, fi);
     SecurityExpressionRoot root = new
       CustomWebSecurityExpressionRoot(authentication, fi);
     ctx.setRootObject(root);
       return ctx;
   }
}
```

Finally, reconfigure bean references when setting up the `Voter` as follows:

```
<bean class="com.packtpub.springsecurity.security
  .CustomWebSecurityExpressionHandler" id="customExpressionHandler"/>
<bean class="org.springframework.security.web.access
  .expression.WebExpressionVoter" id="expressionVoter">
  <property name="expressionHandler" ref="customExpressionHandler"/>
</bean>
```

Now we're able to use the expression to restrict access depending on whether or not the minute on the clock is even or odd.

```
<security:intercept-url pattern="/*" access="evenMinute"/>
```

Obviously, this is a trivial example, but it illustrates the basic steps required to implement a custom SpEL property that you could use to control access to a particular location.
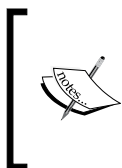
> The technique of configuring a custom SpEL `Voter` may also be used with the `security` namespace configuration by simply defining a custom `AccessDecisionManager` using the `access-decision-manager-ref` attribute, as we saw in Chapter 2.

Summary

In this chapter, we pushed past the standard configuration capability of the Spring Security framework and implemented several advanced customizations. We covered the following:

- Implementing custom servlet filters to handle configurable IP-and role-based filtering, and HTTP request header-based SSO requests

- Adding a custom `AuthenticationProvider` and supporting implementation classes for HTTP request header-based SSO

- Examining the configuration and benefits of session fixation protection and concurrent session handling, including a couple of tangential benefits, which allow for user session reporting

- Configuring custom access denied handling and examining when and why `AccessDeniedException` is thrown, and how it is appropriate to respond to it

- Replicating our auto-configured Spring Security stack by manually declaring all required participants using standard Spring bean XML configuration techniques

- Exploring some advanced capabilities of Spring Bean-based configuration, including session management and event publishing

- Implementing both custom and out of the box `ApplicationListener` to respond to select events raised by the Spring Security framework

- Implementing a custom extension to the standard SpEL expression handler, to allow for customized URL access expressions

What an exciting chapter! We've clearly become very comfortable with the Spring Security framework, and have performed some very advanced extensions and customizations.

In Chapter 7, we'll wrap up our journey through the advanced configuration space when we implement the sophisticated authorization made possible by access control lists.

7
Access Control Lists

In this chapter, we will finally address the complex topic of access control lists, which can provide a rich model of domain object instance-level authorization. Spring Security ships with a robust, but complicated and poorly documented, access control list module which can serve the needs of small to medium-sized implementations reasonably well.

During the course of this chapter we'll:

- Understand the conceptual model of access control lists
- Review the terminology and application of access control list concepts in the Spring Security ACL module
- Build and review the database schema required to support Spring ACL
- Configure JBCP Pets to use ACL secured business methods via annotations and Spring Beans
- Perform advanced configuration, including customized ACL permissions, ACL-enabled JSP tag checks and method security, mutable ACLs, and smart caching
- Examine architectural considerations and plan scenarios for ACL deployment

Using Access Control Lists for business object security

The final piece of the non-web tier security puzzle is security at the business object level, applied at or below the business tier. Security at this level is implemented using a technique known as **access control lists**, or ACLs. Summing up the objective of ACLs in a single sentence—ACLs allow specification of a set of group permissions based on the unique combination of group, business object, and logical operation.

For example, an ACL declaration for JBCP Pets might declare that a given user has write access to their own user profile. This might be shown as:

Username	Group	Object	Permissions
amy		Profile 123	read, write
	ROLE_USER	Profile 123	read
	ANONYMOUS	Any Profile	none

You can see that this ACL is eminently readable by a human—Amy has read and write access to her own customer profile (Profile 123); other registered users can read Amy's profile, but anonymous users cannot. This type of rule matrix is, in a nutshell, what ACL attempts to synthesize into a combination of code, access checking, and metadata about a secured system and its business data. Most true ACL-enabled systems have extremely complex ACL lists, and may conceivably have millions of entries, across the entire system. Although this sounds frighteningly complex, proper up-front reasoning and implementation with a capable security library can make ACL management quite feasible.

If you use a Microsoft Windows or Unix/Linux-based computer, you experience the magic of ACLs every single day. Most modern computer operating systems use ACL directives as part of their file storage systems, allowing permission granting based on a combination of user or group, file or directory, and permission. In Microsoft Windows, you can view some of the ACL capabilities of a file by right-clicking on a file and examining its security properties (**Properties**|**Security** tab):
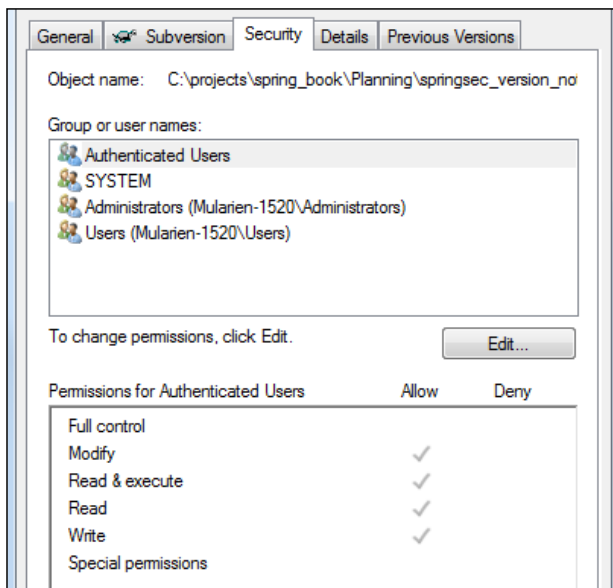
We can see that the combinations of inputs to the ACL are visible and intuitive as you navigate through the various groups or users and permissions.

Access Control Lists in Spring Security

Spring Security supports ACL-driven authorization checks against access to individual domain objects by individual users of the secured system. Much as in the OS file system example, it is possible to use the Spring Security ACL components to build logical tree structures of both business objects and groups or principals. The intersection of permissions (inherited or explicit) on both the requestor and the requestee is used to determine allowed access.

It's quite common for users approaching the ACL capability of Spring Security to be overwhelmed by its complexity, combined with a relative dearth of documentation and examples. This is compounded by the fact that setting up the ACL infrastructure can be quite complicated, with many interdependencies and a reliance on bean-based configuration mechanisms which are quite unlike much of the rest of Spring Security (as you'll see in a moment when we set up the initial configuration).

The Spring Security ACL module was written to be a reasonable baseline, but users intending to build extensively on the functionality will likely run into a series of frustrating limitations and design choices which have gone (for the most part) uncorrected as they were first introduced in the early days of Spring Security. Don't let these limitations discourage you! The ACL module is a powerful way to embed rich access controls in your a pplication, and further scrutinize and secure user actions and data.

Before we dig into configuring Spring Security ACL support, we need to review some key terminology and concepts.

The main unit of secured actor identity in the Spring ACL system is the **security identity**, or **SID**. The SID is a logical construct that can be used to abstract the identity of either an individual principal or a group (`GrantedAuthority`). The SIDs defined by the ACL data model you construct are used as the basis for explicit and derived access control rules when determining the allowed level of access for a particular principal.

If SIDs are used to define actors in the ACL system, the opposite half of the security equation is the definition of the secured objects themselves. The identification of individual secured objects is called (unsurprisingly) an **object identity**. The default Spring ACL implementation of an object identity requires ACL rules to be defined at the individual object instance level that means, if desired, every object in the system can have an individual access rule.

Individual access rules are known as **access control entries**, or **ACE**s. An ACE is the combination of the following factors:

- The SID for the actor to which the rule applies
- The object identity to which the rule applies
- The permission that should be applied to the given SID and the stated object identity
- Whether or not the stated permission should be allowed or denied for the given SID and object identity

The purpose of the Spring ACL system as a whole is to evaluate each secured method invocation and determine whether the object or objects being acted on in the method should be allowed as per the applicable ACEs. Applicable ACEs are evaluated at runtime, based on the caller and the objects in play.

> Spring Security ACL is flexible in its implementation. Although the majority of this chapter details the out of the box functionality of the Spring Security ACL module, keep in mind however, that many of the rules indicated represent default implementations, which in many cases can be overridden based on more complex requirements.

Spring Security uses helpful value objects to represent the data associated with each of these conceptual entities. These are listed in the following table:

ACL Conceptual Object	Java Object
SID	`o.s.s.acls.model.Sid`
Object Identity	`o.s.s.acls.model.ObjectIdentity`
ACL	`o.s.s.acls.model.Acl`
ACE	`o.s.s.acls.model.AccessControlEntry`

Let's work through the process of enabling Spring Security ACL components for a simple demonstration in the JBCP Pets store.

Basic configuration of Spring Security ACL support

Although we hinted previously that configuring ACL support in Spring Security requires bean-based configuration (which it does), you can use ACL support while retaining the simpler `security` XML namespace configuration if you choose. If you're following along with the sample code, you'll need to switch `web.xml` to the `security` namespace configuration, using `dogstore-base.xml` and `dogstore-security.xml`.

Defining a simple target scenario

Our simple target scenario is to disallow read access to the first category on the JBCP Pets home page to any individual other than a user with `ROLE_ADMIN` `GrantedAuthority` assigned. The first category is called **Pet Apparel**—this is the category we'll be protecting with our pass of ACL security.

Although there are several ways to set up ACL checking, our preference is to follow the annotation-based approach that we used in Chapter 5's method level annotations. This nicely abstracts the use of ACLs away from the actual interface declarations, and allows for replacement (if you want) of the role declarations with something other than ACLs at a later date (should you so choose).

We'll add an annotation to the `IProductService.getItemsByCategory` method, which requires that anyone invoking this method has an appropriate role to view the category:

```
@Secured("VOTE_CATEGORY_READ")
public Collection<Item> getItemsByCategory(Category cat);
```

This method is invoked from the category view page of the JBCP Pets site, which is accessible by clicking on a category from the JBCP Pets home page:

Let's get started with our configuration changes!

Adding ACL tables to the HSQL database

The first thing we'll need to do is to add the required tables to support persistent ACL entries into our in-memory HSQL database. To do this, we'll add a new SQL DDL file to our embedded-database declaration in `dogstore-security.xml`:

```
<jdbc:embedded-database id="dataSource" type="HSQL">
  <jdbc:script location="classpath:security-schema.sql"/>
  <jdbc:script location="classpath:test-data.sql"/>
  <jdbc:script location="classpath:remember-me-schema.sql"/>
  <jdbc:script location="classpath:test-users-groups-data.sql"/>
  <jdbc:script location="classpath:acl-schema.sql"/>
</jdbc:embedded-database>
```

The `acl-schema.sql` file will be placed in the `WEB-INF/classes` folder (or another location of your choice on the application classpath), and should contain the following:

```
create table acl_sid (
  id bigint generated by default as identity(start with 100) not null
primary key,
  principal boolean not null,
  sid varchar_ignorecase(100) not null,
  constraint uk_acl_sid unique(sid,principal) );
create table acl_class (
  id bigint generated by default as identity(start with 100) not null
primary key,
  class varchar_ignorecase(500) not null,
  constraint uk_acl_class unique(class) );
create table acl_object_identity (
  id bigint generated by default as identity(start with 100) not null
primary key,
  object_id_class bigint not null,
  object_id_identity bigint not null,
  parent_object bigint,
  owner_sid bigint not null,
  entries_inheriting boolean not null,
  constraint uk_acl_objid unique(object_id_class,object_id_identity),
  constraint fk_acl_obj_parent foreign key(parent_object)references
acl_object_identity(id),
  constraint fk_acl_obj_class foreign key(object_id_class)references
acl_class(id),
  constraint fk_acl_obj_owner foreign key(owner_sid)references acl_
sid(id) );
create table acl_entry (
  id bigint generated by default as identity(start with 100) not null
primary key,
  acl_object_identity bigint not null,
  ace_order int not null,
```

```
    sid bigint not null,
    mask integer not null,
    granting boolean not null,
    audit_success boolean not null,
    audit_failure boolean not null,
    constraint uk_acl_entry unique(acl_object_identity,ace_order),
    constraint fk_acl_entry_obj_id foreign key(acl_object_identity)
        references acl_object_identity(id),
    constraint fk_acl_entry_sid foreign key(sid) references acl_sid(id)
);
```

This will result in the following database schema:
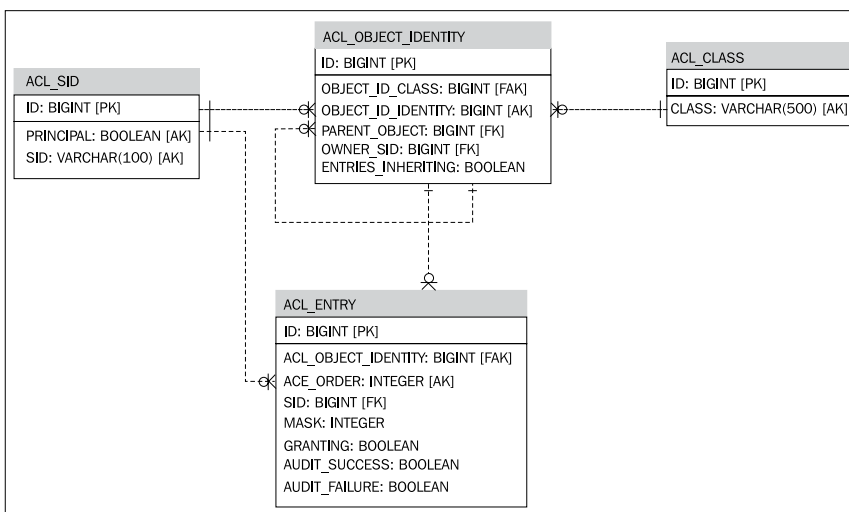
You can see how the concepts of SIDs, object identity, and ACEs map directly to the database schema. Conceptually, this is convenient, as we can map our mental model of the ACL system and how it is enforced directly to the database.

If you've cross referenced this with the HSQL database schema supplied with the Spring Security documentation, you'll note that we've made a few tweaks that commonly bite users. These are as follows:

- Change the ACL_CLASS.CLASS column to 500 characters, from the default value of 100. Some long fully-qualified class names don't fit in 100 characters.
- Name the foreign keys with something meaningful so that failures are more easily diagnosed.

> Note that if you are using another database, such as Oracle, you'll have to translate the DDL into DDL and data types specific to your database.

Once we configure the remainder of the ACL system, we'll return to the database to set up some basic ACEs to prove out the ACL functionality in its most primitive form.

Configuring the Access Decision Manager

We'll need to configure `<global-method-security>` to enable annotations (where we'll annotate based on expected ACL privilege), and reference a custom access decision manager.

> The access decision manager used for ACL checking must be different than the one used for checking web URL authorization rules. This requirement is due to the fact that all authorization checks passed to an access decision manager must be supported by all of its configured voters. Unfortunately, the method of checking web request authorization differs from method request authorization, and as such requires a separately configured access decision manager. When we first visited method security in *Chapter 5*, *Fine-Grained Access Control*, we didn't have to make this explicit configuration change, because the access decision manager instantiated by the `security` namespace parser was sufficient for our needs.

Configuration of the access decision manager reference is set in `dogstore-security.xml` as follows:

```
<global-method-security secured-annotations="enabled"
access-decision-manager-ref="aclDecisionManager"/>
```

This is a bean reference to the access decision manager bean, which we'll define in `dogstore-base.xml`:

```
<bean class="org.springframework.security.access.vote.
AffirmativeBased" id="aclDecisionManager">
  <property name="decisionVoters">
    <list>
      <ref bean="categoryReadVoter"/>
    </list>
  </property>
</bean>
```

We'll resolve the reference chain starting with the voter in the next step of the exercise. Remember that this is an `AccessDecisionManager` just like any other, and, just as the web authorization decision manager does, relies upon voters to make authorization decisions.

Configuring supporting ACL beans

With even a relatively straightforward ACL configuration, as we have in our scenario, there are a number of required dependencies to set up. As we mentioned previously, the Spring Security ACL module comes out of the box with a number of components that you can assemble to provide a decent set of ACL capabilities. Take note that all of the components we'll reference in the following diagram are part of the framework!
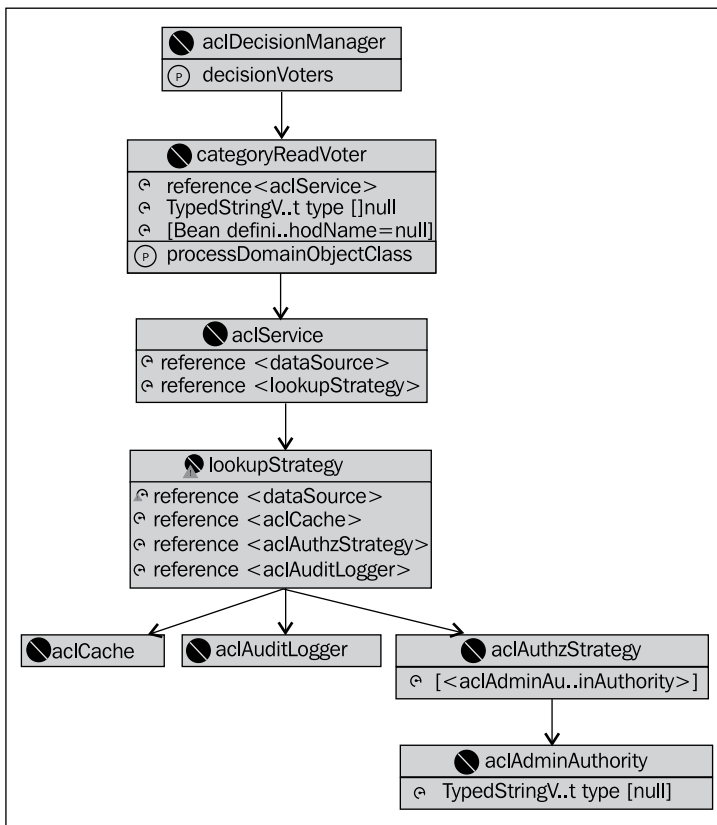
One obvious difference between the ACL subsystem and the majority of the rest of the Spring Security framework is that most of the configuration requires use of **constructor injection**, instead of property injection. As we discussed with explicit bean configuration in *Chapter 6, Advanced Configuration and Extension*, portions of Spring Security are inconsistent about the forced use of constructors to ensure that the required properties are set—just be aware of this, when you are configuring most of the ACL-related components.

Let's begin first with configuring the `categoryReadVoter`. This is the `AccessDecisionVoter` implementation that will consult the ACL repository (in the database) and runtime authorization and make an access decision.

```
<bean class="org.springframework.security.acls.AclEntryVoter"
id="categoryReadVoter">
  <constructor-arg ref="aclService"/>
  <constructor-arg value="VOTE_CATEGORY_READ"/>
  <constructor-arg>
    <array>
      <util:constant static-field="org.springframework.security.acls.
domain.BasePermission.READ"/>
    </array>
  </constructor-arg>
  <property name="processDomainObjectClass" value="com.packtpub.
springsecurity.data.Category"/>
</bean>
```

The attributes that are not obvious bean references bear some explanation. The second constructor argument, for which we've provided a value of `VOTE_CATEGORY_READ`, represents the name of the security attribute that this voter can vote on. You may remember that this matches the `@Secured` annotation on the method that we'll be protecting with ACL.

The third constructor argument and property name combine to effectively declare the ACL properties that would be expected in order to confer a successful vote on a requestor. In this case, the declaration of the voter states that in order to be authorized for a method requiring `VOTE_CATEGORY_READ` access, the requestor must have an ACL entry indicating that they are allowed `READ` access on the `com.packtpub.springsecurity.data.Category` domain object.

We can see that the declaration of the ACL voter is the layer of abstraction between authorization requirements declared in code resources and the assignment of permissions on domain objects to ACL SIDs themselves. This layer of abstraction allows the utmost flexibility in assignment of meaningful security properties to secured resources.

The bean reference to `aclService` resolves to an implementation of `o.s.s.acls.model.AclService` that is responsible (through delegation) for translating information about the object being secured by ACLs into expected ACEs.

```
<bean class="org.springframework.security.acls.jdbc.JdbcAclService"
id="aclService">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="lookupStrategy"/>
</bean>
```

We'll use `o.s.s.acls.jdbc.JdbcAclService`, which is the implementation of `AclService`. This implementation comes out of the box and is ready to use the schema that we defined in the last step of this exercise. `JdbcAclService` will additionally use recursive SQL and post-processing to understand object and SID hierarchies, and ensure representations of these hierarchies are passed back to the `AclEntryVoter`.

The `JdbcAclService` uses the same JDBC `dataSource` that we've defined with the embedded-database declaration, and also delegates to an implementation of `o.s.s.acls.jdbc.LookupStrategy`, which is solely responsible for actually making database queries and resolving requests for ACLs. The only `LookupStrategy` supplied with Spring Security is `o.s.s.acls.jdbc.BasicLookupStrategy`, defined as follows:

```
<bean class="org.springframework.security.acls.jdbc.
BasicLookupStrategy" id="lookupStrategy">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="aclCache"/>
  <constructor-arg ref="aclAuthzStrategy"/>
  <constructor-arg ref="aclAuditLogger"/>
</bean>
```

Now, `BasicLookupStrategy` is a relatively complex beast. Remember that its purpose is to translate a list of `ObjectIdentity`s to be protected into the actual, applicable ACE list from the database. As `ObjectIdentity` declarations can be recursive, this proves to be quite a challenging problem, and a system which is likely to experience heavy use should consider the SQL that's generated for performance impact on the database.

Querying with the lowest common denominator

Be aware that `BasicLookupStrategy` is intended to be compatible with all databases by strictly sticking with standard ANSI SQL syntax, notably left [outer] joins. Some older databases (notably, Oracle 8i) did not support this join syntax, so be sure to verify that the syntax and structure of SQL is compatible with your particular database!

There are also most certainly more efficient database-dependent methods of performing hierarchical queries using non-standard SQL—for example, Oracle's `CONNECT BY` statement and the **Common Table Expression** (**CTE**) capability of many other databases, including PostgreSQL and Microsoft SQL Server.

Much as we learned in Chapter 4's example of using a custom schema for the `JdbcDaoImpl UserDetailsService`, properties are exposed to allow for configuration of the SQL utilized by `BasicLookupStrategy`. Please consult the Javadoc and the source code itself to see how they are used, so that they can be correctly applied to your custom schema.

We can see that the `LookupStrategy` requires a reference to the same JDBC `dataSource` that the `AclService` utilizes. The other three references bring us almost to the end of the dependency chain.

`o.s.s.acls.model.AclCache` declares an interface for a caching `ObjectIdentity` to `Acl` mappings, to prevent redundant (and expensive) database lookups. Spring Security ships with only one implementation of `AclCache`, using the third-party library Ehcache. For simplicity at this point in the configuration, we'll avoid the additional work of configuring Ehcache, and instead implement a simple, custom no-op `AclCache`, in the class `com.packtpub.springsecurity.security.NullAclCache`:

```
package com.packtpub.springsecurity.security;
// imports omitted
public class NullAclCache implements AclCache {
  @Override
  public void clearCache() {   }
  @Override
  public void evictFromCache(Serializable arg0) { }
  @Override
  public void evictFromCache(ObjectIdentity arg0) {        }
  @Override
  public MutableAcl getFromCache(ObjectIdentity arg0) {
    return null;
  }
  @Override
  public MutableAcl getFromCache(Serializable arg0) {
    return null;
  }
  @Override
  public void putInCache(MutableAcl arg0) { }
}
```

This is configured through a simple bean declaration:

```
<bean class="com.packtpub.springsecurity.security.NullAclCache"
id="aclCache"/>
```

Don't worry, we will configure the Ehcache-based implementation later in this chapter, but for now we want to focus on configuring the absolutely required ACL components first!

The next simple dependency hanging off of `BasicLookupStrategy` is an implementation of the `o.s.s.acls.domain.AuditLogger` interface, which is used by the `BasicLookupStrategy` to audit ACL and ACE lookups. Similar to the `AclCache` interface, only one implementation is supplied with Spring Security which simply logs to the console. We'll configure it with another one-line bean declaration:

```
<bean class="org.springframework.security.acls.domain.
ConsoleAuditLogger" id="aclAuditLogger"/>
```

The final dependency to resolve is to an implementation of the `o.s.s.acls. domain.AclAuthorizationStrategy` interface, which actually has no immediate responsibility during the load of the ACL from the database at all. Instead, the implementation of this interface is responsible for determining whether a runtime change to an ACL or ACE is allowed, based on the type of change. We'll explain more on this later when we cover mutable ACLs, as the logical flow is both somewhat complicated and not pertinent to getting our initial configuration complete. The final configuration requirements are as follows:

```
<bean class="org.springframework.security.acls.domain.
AclAuthorizationStrategyImpl" id="aclAuthzStrategy">
  <constructor-arg>
    <array>
      <ref local="aclAdminAuthority"/>
      <ref local="aclAdminAuthority"/>
      <ref local="aclAdminAuthority"/>
    </array>
  </constructor-arg>
</bean>
<bean class="org.springframework.security.core.authority.
GrantedAuthorityImpl" id="aclAdminAuthority">
  <constructor-arg value="ROLE_ADMIN"/>
</bean>
```

You might wonder what the repeating references to the `aclAdminAuthority` are for—the `AclAuthorizationStrategyImpl` provides three specific `GrantedAuthority` designations to allow specific operations at runtime on mutable ACLs. We'll cover these later in the chapter.

We're finally done with basic configuration of an out of the box Spring Security ACL implementation. The next and final step requires that we insert a simple ACL and ACE into the HSQL database, and test it out!

Creating a simple ACL entry

Recall that our very simple scenario is to lock down the first category in the JBCP Pets store so that no one other than users with ROLE_ADMIN authorization can view it. You may find it helpful to refer back several pages to the database schema diagram to follow which data we are inserting and why.

Create a file, test-acl-data.sql in WEB-INF/classes, side by side with the other SQL files that we're using in JBCP Pets. All the SQL explained in this section will be added to this file — you may feel free to experiment and add more test cases based on the sample SQL we've provided — in fact, we'd encourage that you experiment with sample data!

First we'll need to populate the ACL_CLASS table with any or all of the domain object classes which may have ACL rules — in the case of our example, this is simply our Category class:

```
insert into acl_class (class) values ('com.packtpub.springsecurity.
data.Category');
```

Next, the ACL_SID table is seeded with any SIDs, which will be tied to ACEs. Remember that SIDs can either be roles or users — we'll simply populate the roles in our application here (note that the principal column indicates if a given entry is an individual principal or not):

```
insert into acl_sid (principal, sid) values (false, 'ROLE_USER');
insert into acl_sid (principal, sid) values (false, 'ROLE_ADMIN');
```

The table where things start getting complicated is the ACL_OBJECT_IDENTITY table that is used to declare individual domain object instances and their parent (if any) and owning SID. We'll insert a row with the following properties:

- Domain Object of type Category (FK to ACL_CLASS via OBJECT_ID_CLASS column).
- Domain Object PK of 1 (OBJECT_ID_IDENTITY column).
- Owner SID of ROLE_ADMIN (FK to ACL_SID via OWNER_SID column).

The SQL to insert a row here, for the Category with a PK of 1, is:

```
insert into acl_object_identity (object_id_class,object_id_
identity,parent_object,owner_sid,entries_inheriting)
select cl.id, 1, null, sid.id, false
from acl_class cl, acl_sid sid
where cl.class='com.packtpub.springsecurity.data.Category' and sid.
sid='ROLE_ADMIN';
```

Keep in mind that in a typical scenario, the owning SID would represent a principal, and not a role; however, both types of rules function equally well as far as the ACL system is concerned.

Finally, we'll add an ACE related to this object instance which declares that the ROLE_ADMIN role is allowed read access to the object:

```
insert into acl_entry (acl_object_identity, ace_order, sid, mask,
granting, audit_success, audit_failure)
select oi.id, 1, si.id, 1, true, true, true
from acl_object_identity oi, acl_sid si
where si.sid = 'ROLE_ADMIN';
```

The MASK column here represents a bitmask which is used to grant permission assigned to the stated SID on the object in question. We'll explain the details of this later in this chapter—fortunately, it doesn't tend to be as useful as it may sound.

With the SQL file now populated, we can augment our `<embedded-database>` declaration and add the final test ACL data file to the database bootstrap:

```
<jdbc:embedded-database id="dataSource" type="HSQL">
<!--additional SQL files omitted -->
    <jdbc:script location="classpath:acl-schema.sql"/>
    <jdbc:script location="classpath:test-acl-data.sql"/>
</jdbc:embedded-database>
```

Now we can start the application and run through our sample scenario. You should see that when any user other than the administrative user attempts to access the first category, **Pet Apparel**, they are denied access. Note also that if unauthenticated users attempt to access this category, they follow the standard AccessDeniedException handling (described in Chapter 6) and are required to log in.

We now have a basic working setup of ACL-based security (albeit, a very simple scenario). Let's move on to some more explanation about concepts we saw during this walkthrough, and then review a couple of considerations in a typical Spring ACL implementation that you should consider before using it.

Advanced ACL topics

Some high-level topics that we skimmed over during the configuration of our ACL environment had to do with ACE permissions, and the use of GrantedAuthority indicators to assist the ACL environment in determining whether certain types of runtime changes to ACLs were allowed. Now that we have a working environment, we'll review these more advanced topics.

How permissions work

Permissions are no more than single logical identifiers represented by bits in an integer. An access control entry grants permissions to SIDs based on the bitmask which comprises the logical ANDing of all permissions applicable to that access control entry.

The default `Permission` implementation, `o.s.s.acls.domain.BasePermission` defines a series of integer values representing common ACL authorization verbs. These integer values correspond to single bits set in an integer, so a value of BasePermission. WRITE, with integer value `1`, has a bitwise value of 2^1 or `2`. These are illustrated in the following diagram:
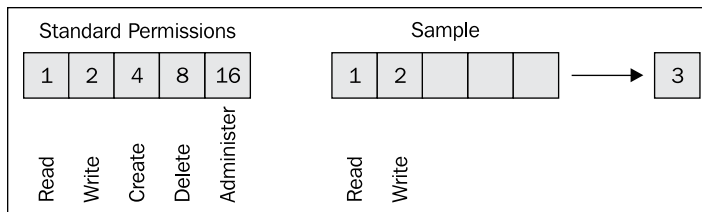
We can see that the sample permission bitmask would have an integer value of 3, due to the application of both the Read and Write permissions to the permission value. All of the standard integer single permission values shown in the diagram are defined in the `BasePermission` object as static constants. You may recall that we used one of these constants, `BasePermission.READ`, when we constructed our `o.s.s.acls.AclEntryVoter` in our ACL configuration exercise.

> The logical constants that are included in `BasePermission` are just a sensible baseline of commonly used permissions in access control entries, and have no semantic meaning within the Spring Security Framework. It's quite common for very complex ACL implementations to invent their own custom permissions, augmenting best practice examples with domain-or business-dependent ones.

One issue that often confuses users is how the bitmasks are used in practice, given that many databases either do not support bitwise logic, or do not support it in a scalable way. Spring ACL intends to solve this problem by putting more of the load of calculating appropriate permissions vis-à-vis bitmasks on the application, rather than on the database.

It's important to review the resolution process, where we see how the `AclEntryVoter` resolves permissions declared on the method itself (in our example, with the `@Secured` annotation) to real ACL permissions. The following diagram illustrates the process that Spring ACL performs to evaluate the declared permission against the relevant ACEs for the requesting principal:
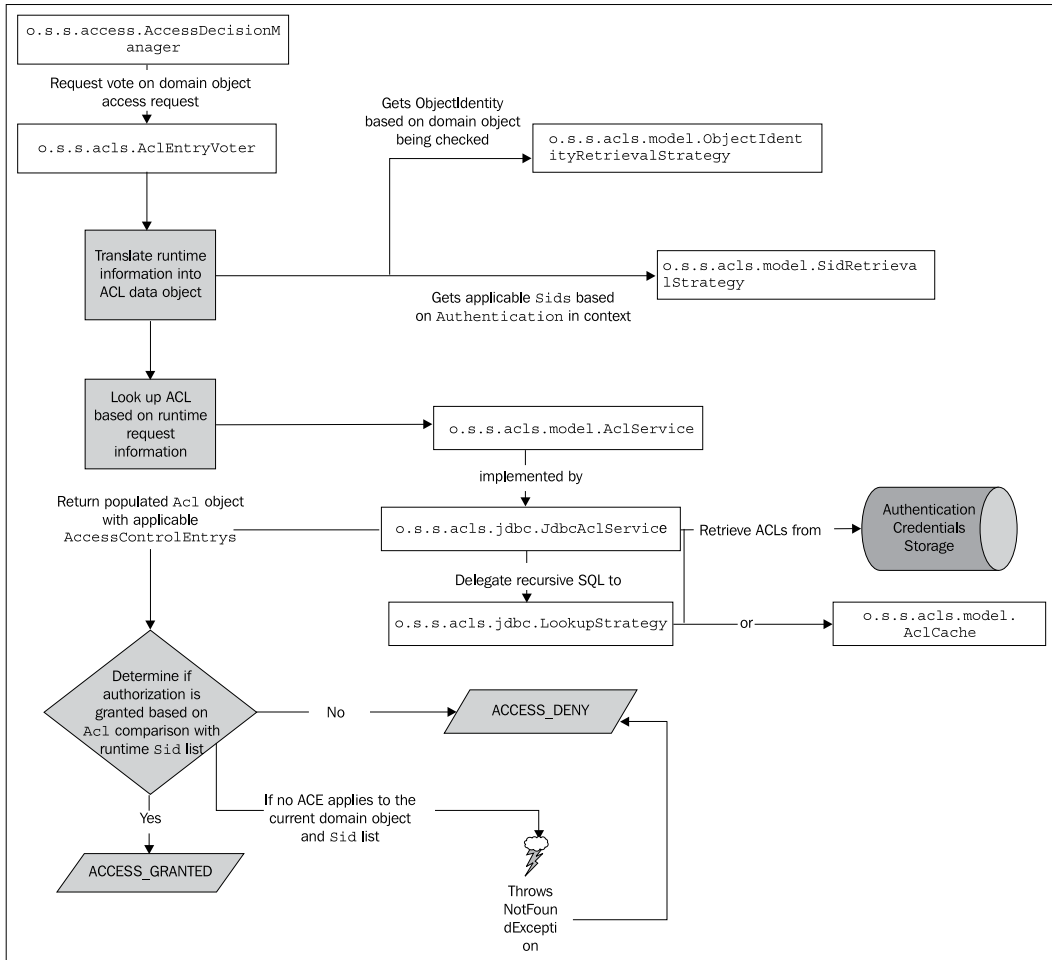
We see that the `AclEntryVoter` relies on classes implementing two interfaces, `o.s.s.acls.model.ObjectIdentityRetrievalStrategy`, and `o.s.s.acls.model.SidRetrievalStrategy`, to retrieve the `ObjectIdentity` and `Sids` appropriate for the authorization check. The important thing to note about these strategies is how the default implementation classes actually determine the `ObjectIdentity` and `Sids` to return, based on the context of the authorization check.

`ObjectIdentity` has two properties, `type` and `identifier`, that are derived from the object being checked at runtime, and used to declare ACE entries. The default `ObjectIdentityRetrievalStrategy` uses the fully-qualified class name to populate the `type` property. The `identifier` property is populated with the result of a method with the signature `Serializable getId()`, invoked on the actual object instance.

> As your object isn't required to implement an interface to be compatible with ACL checks, the requirement to implement a method with a specific signature can be surprising for developers implementing Spring Security ACL. Plan ahead, and ensure that your domain objects contain this method! You may also implement your own `ObjectRetrievalStrategy` (or subclass the out of the box implementation) to call a method of your choice. The name and type signature of the method is, unfortunately, not configurable.

Unfortunately, the actual implementation of `AclImpl` directly compares the `Permission` configured with the `AclEntryVoter`, and the Permission stored on the ACE in the database, without using bitwise logic. The Spring Security community is in debate about whether this is unintentional or working as intended, but regardless, you will need to take care when declaring a user with a combination of permissions, as either the `AclEntryVoter` must be configured with all combinations of permission, or the ACEs need to ignore the fact that the permission field is intended to store multiple values, and instead store a single permission per ACE.

If you want to verify this with our simple scenario, change the Read permission we granted to the ROLE_ADMIN SID to the bitmask combination of Read and Write, which translates to a value of `3`. This would be changed in `test-acl-data.sql`:

```
insert into acl_entry (acl_object_identity, ace_order, sid, mask,
granting, audit_success, audit_failure)
select oi.id, 1, si.id, 3, true, true, true
from acl_object_identity oi, acl_sid si
where si.sid = 'ROLE_ADMIN';
```

You can see that if you now try to access the ACL protected `Category` as the `admin` user, you'll be denied, even though we've declared that you have both Read and Write access in a single ACE.

Custom ACL permission declaration

As stated in the earlier discussion on permission declarations, permissions are nothing but logical names for integer bit values. As such, it's possible to extend the `BasePermission` class and declare your own permissions. We'll cover a very straightforward scenario here, where we create a new ACL permission called `ADMIN_READ`. This is a permission that will be granted only to administrative users, and will be assigned to protect resources only administrators could read. Although a contrived example for the JBCP Pets site, this type of use of custom permissions occurs quite often in situations dealing with personally identifiable information (for example, social security number, and so on—recall that we covered PII in *Chapter 1, Anatomy of an Unsafe Application*).

Let's get started making the changes required to support this. The first step is to extend `BasePermission` with our own `com.packtpub.springsecurity.security.CustomPermission` class:

```
package com.packtpub.springsecurity.security;
// imports omitted
public class CustomPermission extends BasePermission {
  protected CustomPermission(int mask, char code) {
    super(mask, code);
  }
  protected CustomPermission(int mask) {
    super(mask);
  }
    public static final Permission ADMIN_READ = new CustomPermission(1
<< 5, 'M'); // 32
}
```

Next, we will need to extend the `o.s.s.acls.domain.PermissionFactory` default implementation, `o.s.s.acls.domain.DefaultPermissionFactory`, to register our custom permission logical value. The role of the `PermissionFactory` is to resolve permission bitmasks into logical permission values (which can be referenced by constant value, or by name, such as `ADMIN_READ`, in other areas of the application). The `PermissionFactory` requires that any custom permissions be registered with it for proper lookup.

We'll implement the `com.packtpub.springsecurity.security.CustomPermissionFactory` class as follows:

```
package com.packtpub.springsecurity.security;
// imports omitted
public class CustomPermissionFactory extends DefaultPermissionFactory
{
  public CustomPermissionFactory() {
```

```
    super();
    registerPublicPermissions(CustomPermission.class);
  }
  public CustomPermissionFactory(Class<? extends Permission>
permissionClass) {
    super(permissionClass);
  }
  public CustomPermissionFactory(
      Map<String, ? extends Permission> namedPermissions) {
    super(namedPermissions);
  }
}
```

We can see that we augment the default constructor with a call to register our CustomPermission class as an available permission.

> We won't highlight all of the code available in the base class as part of the exercises in this chapter, but we'd definitely suggest checking out the code for the superclass to see what other functionality is there, and how it's used in other aspects of the ACL system. For example, we'll see that the buildFromName method will be used in the ACL-enabled JSP custom tag which shows up later in this chapter.

We'll need to configure our CustomPermissionFactory and wire it into the BasicLookupStrategy. We'll make the following changes in dogstore-base.xml:

```
<bean class="org.springframework.security.acls.jdbc.
BasicLookupStrategy" id="lookupStrategy">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="aclCache"/>
  <constructor-arg ref="aclAuthzStrategy"/>
  <constructor-arg ref="aclAuditLogger"/>
  <property name="permissionFactory" ref="customPermissionFactory"/>
</bean>
<bean class="com.packtpub.springsecurity.security.
CustomPermissionFactory" id="customPermissionFactory"/>
```

Now, our custom ACL permission will be available for use in the ACL framework. Next, we'll add a new administrative user who has been explicitly assigned this permission to secure the second category, **Dog Food**. We'll add the following to test-acl-data.sql to take care of this new authorization requirement:

```
-- User SID
insert into acl_sid (principal, sid) values (true, 'admin2');
-- Category #2
```

```
insert into acl_object_identity (object_id_class,object_id_
identity,parent_object,owner_sid,entries_inheriting)
select cl.id, 2, null, sid.id, false
from acl_class cl, acl_sid sid
where cl.class='com.packtpub.springsecurity.data.Category' and sid.
sid='admin2';

-- Give user 'admin2' access to category 2
-- "32" == 1 << 5
insert into acl_entry (acl_object_identity, ace_order, sid, mask,
granting, audit_success, audit_failure)
select oi.id, 2, si.id, 32, true, true, true
from acl_object_identity oi, acl_sid si
where si.sid = 'admin2' and oi.object_id_identity = 2;
commit;
```

We can see that the new integer bitmask value of 32 has been referenced in the ACE data — this intentionally corresponds to our new ADMIN_READ ACL permission as defined in Java code. The **Dog Food** category is referenced by its primary key (stored in the object_id_identity column) value of 2, in the ACL_OBJECT_IDENTITY table.

We'll also need to declare a new AclEntryVoter in the dogstore-base.xml file:

```
<bean class="org.springframework.security.acls.AclEntryVoter" id="admi
nResourceReadVoter">
  <constructor-arg ref="aclService"/>
  <constructor-arg value="VOTE_ADMIN_READ"/>
  <constructor-arg>
    <array>
      <util:constant static-field="com.packtpub.springsecurity.
security.CustomPermission.ADMIN_READ"/>
    </array>
  </constructor-arg>
  <property name="processDomainObjectClass" value="com.packtpub.
springsecurity.data.Category"/>
</bean>
```

Additionally, we'll add this voter to the access decision manager responsible for driving authorization decisions in our ACL protected method scenario:

```
<bean class="org.springframework.security.access.vote.
AffirmativeBased" id="aclDecisionManager">
  property name="decisionVoters">
    <list>
      <ref bean="categoryReadVoter"/>
      <ref bean="adminResourceReadVoter"/>
```

```
      </list>
    </property>
  </bean>
```

And finally, we'll add the required role to the method declaration itself, in the
`IProductService` interface declaration:

```
public interface IProductService {
  // other methods omitted
  @Secured({"VOTE_CATEGORY_READ","VOTE_ADMIN_READ"})
  public Collection<Item> getItemsByCategory(Category cat);
}
```

With all these configurations in place, we can start up the site again and test out the
custom ACL permission. Based on the sample data we have configured, here is what
should happen when the various available users click on categories:

Username	Pet apparel (Category 1)	Dog Food (Category 2)	Any Other category
admin	Allowed (has READ permission via ROLE_ADMIN SID ACE)	Denied	Allowed
admin2	Allowed (has READ permission via ROLE_ADMIN SID ACE)	Allowed (has ADMIN_READ permission via principal SID ACE)	Allowed
guest	Denied	Denied	Allowed

We can see that even with the use of our simple cases, we've now been able to extend
the Spring ACL functionality in a very limited way to illustrate the power of this
fine grained access control system, solely based on the combination of principal,
`GrantedAuthority`, individual domain objects, and business methods.

ACL-Enabling your JSPs with the Spring Security JSP tag library

We saw in *Chapter 3, Enhancing the User Experience* and Chapter 5 that the Spring
Security JSP tag library offers functionality to expose authentication-related data
to the user, and to restrict what the user can see based on a variety of rules.

The very same tag library can also interact with an ACL-enabled system right out of the box! From our simple experiments above, we have configured a simple ACL authorization scenario around the first two categories in the list on the home page.

Let's take this one step further, and use the `<accesscontrollist>` tag to hide the categories that the user doesn't actually have access to.

Please refer to the table in the previous section as a refresher of the access rules we've configured up to this point!

We'll wrap the display of each category with the `<accesscontrollist>` tag, declaring the list of permissions to check on the object to be displayed:

```
<c:forEach var="category" items="${categories}">
<security:accesscontrollist hasPermission="READ,ADMIN_READ" domainObje
ct="${category}">
    <li><a href="category.do?id=${category.name}">${category.name}
</a></li>
</security:accesscontrollist>
</c:forEach>
```

Think for a moment about what we want to occur here—we want the user to see only the items to which they actually have READ or ADMIN_READ (our custom permission) access. We therefore declare the list of permissions to check as a comma-delimited list, and the domain object on which to perform the check (specified with the JSP EL expression `${category}`).

Behind the scenes, the tag implementation utilizes the same `SidRetrievalStrategy` and `ObjectIdentityRetrievalStrategy` discussed earlier in this chapter, so the computation of access checking follows the same workflow as it does with ACL-enabled voting on method security.

Spring Expression Language support for ACLs

SpEL support for an ACL-enabled system is allowed for method security only, with the use of the `hasPermission` SpEL function. Typically, this type of access check is combined with a reference to one or more parameters to the function (for `@PreAuthorize` checks), or for collection filtering (with `@PostAuthorize` checks).

Unfortunately, the configuration requirements for enabling ACL support in method security require us to configure all of method security explicitly using Spring Bean configuration. As such, we will need to remove the `<global-method-security>` element and replace it with the explicit method security configuration we covered at the end of Chapter 6. While we won't repeat the configuration here (although it is included in the code for this chapter), we do need to make the following minor changes so that the required support classes for the ACL `hasPermission` check are enabled:

```
<bean class="org.springframework.security.access.expression.method.
DefaultMethodSecurityExpressionHandler" id="methodExprHandler">
  <property name="permissionEvaluator" ref="aclPermissionEvaluator"/>
</bean>
<bean class="org.springframework.security.acls.AclPermissionEvaluator"
id="aclPermissionEvaluator">
  <constructor-arg ref="aclService"/>
  <property name="permissionFactory" ref="customPermissionFactory"/>
</bean>
```

We have updated the `methodExprHandler` bean definition to configure a `o.s.s.access.PermissionEvaluator` implementation (the default `PermissionEvaluator` implementation simply denies all permission checks). The `o.s.s.acls.AclPermissionEvaluator` utilizes the `AclService` and related classes to actually check the permission declared in the SpEL `hasPermission` expression.

With this configuration in place, we can use an alternate method to filter the list of categories displayed on the home page (make sure you remove the tag library additions we made in the prior exercise!).
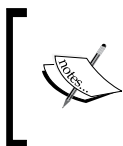
> Take note that the `hasPermission` function does not support a comma-separated list of permissions (as we saw with the `<accesscontrollist>` JSP tag), so we need to use SpEL boolean logic.

Simply add the following declaration to the `IProductService` interface.

```
@PostFilter("hasPermission(filterObject, 'READ') or hasPermission(filterObject, 'ADMIN_READ')")
Collection<Category> getCategories();
```

Now, try restarting the application, and compare the list of categories shown on the home page when logged in anonymously, as the `admin` user, and as the `admin2` user. Notice how the displayed list is completely driven by ACL permissions? We can see that the `hasPermission` SpEL function will nicely tie our ACL-enabled application with method-level security annotations!

Mutable ACLs and authorization

Although the JBCP Pets site doesn't implement full user administration functionality, it's likely that your application will have common features such as new user registration and administrative user maintenance. To this point, lack of these features—which we have worked around using SQL inserts at application startup—hasn't stopped us from demonstrating many of the features of Spring Security and Spring ACL.

However, the proper handling of runtime changes to declared ACLs, or the addition or removal of users in the system, is critical to maintaining the consistency and security of the ACL-based authorization environment. Spring ACL solves this issue through the concept of the mutable ACL (`o.s.s.acls.model.MutableAcl`).

Extending the standard `Acl` interface, the `MutableAcl` allows for runtime manipulation of ACL fields in order to change the in-memory representation of a particular ACL. This additional functionality includes the ability to create, update, or delete ACEs, change ACL ownership, and other useful functions.

We might expect, then, that the Spring ACL module would come out of the box with a way to persist runtime ACL changes to the JDBC data store, and indeed it does. The `o.s.s.acls.jdbc.JdbcMutableAclService` may be used to create, update, and delete `MutableAcl` instances in the database, as well as to do general maintenance on the other supporting tables for ACLs (handling SIDs, `ObjectIdentity`, and domain object class names).

It takes only a slight configuration change for us to use the `JdbcMutableAclService` instead of the `JdbcAclService`—the mutable service requires a reference to the ACL cache, so that it can appropriately evict items from the cache when they are updated in the database. The bean configuration is appropriately straightforward:

```
<bean class="org.springframework.security.acls.jdbc.
JdbcMutableAclService" id="mutableAclService">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="lookupStrategy"/>
  <constructor-arg ref="aclCache"/>
</bean>
```

Recall from earlier in the chapter that the `AclAuthorizationStrategyImpl` allows us to specify certain roles required for actions on mutable ACLs. These are supplied to the constructor as part of the bean configuration. The constructor arguments, and their meanings, are as follows:

Arg #	What it does
1	Indicates the authority a principal is required to have in order to take ownership of an ACL-protected object at runtime.
2	Indicates the authority a principal is required to have in order to change the auditing of an ACL-protected object at runtime.
3	Indicates the authority a principal is required to have in order to make any other kind of change (create, update, and delete) to an ACL-protected object at runtime.

The `JdbcMutableAclService` contains a number of methods used to manipulate ACL and ACE data at runtime. While the methods themselves are fairly understandable (`createAcl`, `updateAcl`, `deleteAcl`), the correct way to configure and use the `JdbcMutableAclService` is often difficult for even advanced Spring Security users.

Let's implement an ACL bootstrap class that will replace the need for our bootstrap SQL script, and instead insert our current ACL and ACE entries (and supporting `ObjectIdentity`s and `Sid`s) programmatically.

Configuring a Spring transaction manager

The `JdbcMutableAclService` uses Spring's `JdbcTemplate` support to interact with a JDBC DataSource. As such, it requires the presence of a Spring JDBC `PlatformTransactionManager`, and verifies (very aggressively) that all interactions with the database are properly wrapped in transactions.

While most real applications using Spring will probably already have a transaction manager declared, our JBCP Pets application does not. We'll simply add one in `dogstore-base.xml`:

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

We'll declare a reference to the bootstrap bean that we'll be coding in a moment:

```
<bean class="com.packtpub.springsecurity.security.AclBootstrapBean"
init-method="aclBootstrap"/>
```

Much like our `DatabasePasswordSecurerBean` from *Chapter 4*, *Securing Credential Storage*, we've configured this bean so that the `aclBootstrap` method will be invoked when the Spring `ApplicationContext` is initialized—the perfect time to bootstrap our application's ACL data!

Interacting with the JdbcMutableAclService

Now, we'll code up our bootstrap bean—create the `com.packtpub.springsecurity.security.AclBootstrapBean` class. First, we'll use `@Autowired` to inject the required dependencies:

```
package com.packtpub.springsecurity.security;
// imports omitted
public class AclBootstrapBean {
  @Autowired
  MutableAclService mutableAclService;
  @Autowired
  IProductDao productDao;
  @Autowired
  PlatformTransactionManager transactionManager;
```

Next, the actual definition of the method, which will be invoked to bootstrap the ACL data—we'll analyze this one segment at a time.

```
public void aclBootstrap() {

// domain data to set up
Collection<Category> categories = productDao.getCategories();
Iterator<Category> iterator = categories.iterator();
final Category category1 = iterator.next();
final Category category2 = iterator.next();
```

We'll need references to the actual domain objects to be secured in order to create ACLs for them. Recall that our ACL bootstrap SQL adds entries for only the first two categories in our store, so we retrieve them from the Product DAO.

The `JdbcMutableAclService` will use an `ObjectIdentity` to create the initial `MutableAcl` (which can then be further manipulated, as we will see). To create an `ObjectIdentity`, we will need the actual domain object.

```
// needed because MutableAclService requires a current authenticated
principal
GrantedAuthorityImpl roleUser = new GrantedAuthorityImpl("ROLE_USER");
GrantedAuthorityImpl roleAdmin = new GrantedAuthorityImpl("ROLE_
ADMIN");
UsernamePasswordAuthenticationToken token = new UsernamePasswordAuthen
ticationToken("admin","admin",Arrays.asList(new GrantedAuthority[]
{roleUser, roleAdmin}));
SecurityContextHolder.getContext().setAuthentication(token);
```

The `JdbcMutableAclService` verifies that a user has been logged in, and uses this logged-in user as the default owner for the created `MutableAcl`. Unfortunately, this is a mandatory check, even if you are later intending to explicitly set the ACL's owner. As this code is executing without an authenticated user, it fools the `JdbcMutableAclService` into thinking that the `admin` user is currently authenticated.

```
// sids
final Sid userRole = new GrantedAuthoritySid("ROLE_USER");
final Sid adminRole = new GrantedAuthoritySid("ROLE_ADMIN");
// users
final Sid adminUser = new PrincipalSid("admin");
final Sid admin2User = new PrincipalSid("admin2");
```

We need to create `Sid`s for the principals and groups which will be represented within the ACL and ACE structure, so we explicitly create them here. Keep in mind that if you were manipulating ACLs within the scope of the application (for example, in a business service invoked from the UI layer), you might approach this problem in a different way—the code here is intended only as a working example, which you are encouraged to adapt to your situation.

```
// all interaction with JdbcMutableAclService must be within a
transaction
TransactionTemplate tt = new TransactionTemplate(transactionManager);
  tt.execute(new TransactionCallbackWithoutResult() {
  @Override
  protected void doInTransactionWithoutResult(TransactionStatus arg0)
{
  // category 1 ACL
  MutableAcl createAclCategory1 = mutableAclService.createAcl(new Obje
ctIdentityImpl(category1));
  createAclCategory1.setOwner(adminRole);
  createAclCategory1.insertAce(0, BasePermission.READ, adminRole,
true);
  mutableAclService.updateAcl(createAclCategory1);

  // category 2 ACL
  MutableAcl createAclCategory2 = mutableAclService.createAcl(new Obje
ctIdentityImpl(category2));
  createAclCategory2.setOwner(admin2User);
  createAclCategory2.insertAce(0, CustomPermission.ADMIN_READ,
admin2User, true);
  mutableAclService.updateAcl(createAclCategory2);
}});

SecurityContextHolder.clearContext();
}
}
```

Within the scope of a new database transaction, the interaction with the `JdbcMutableAclService` is performed. We can see that the initial `createAcl` call returns a `MutableAcl` object. Behind the scenes, database inserts or lookups are performed for the `ObjectIdentity` and `Sid`. The `MutableAcl` itself provides methods for creating, updating, or deleting ACEs on the ACL (remember, ACEs declare individual permission-to-SID mappings). Finally, the `MutableAcl` is updated in the database with the `updateAcl` method call.

Keep in mind that the `JdbcMutableAclService` is also responsible for ensuring the `AclCache` is refreshed as `MutableAcl` operations are performed!

You are encouraged to experiment with the mutable ACL service and enhance the site to support user addition and editing—the `JdbcMutableAclService` is quite well documented (at the code level), and attempting to implement it will be a good exercise in determining your readiness to fully embrace a runtime-driven ACL model.

Ehcache ACL caching

Ehcache is an open-source, memory and disk-based caching library which is widely used in many open-source and commercial Java products. As mentioned earlier in the chapter, Spring Security ships with a default implementation of ACL caching, which relies on the availability of a configured Ehcache instance, which it uses to store ACL information in preference to reading ACLs from the database.

While deep configuration of Ehcache is not something we want to cover in this section, we'll cover how Spring ACL uses the cache, and walk you through a basic, default configuration.

Configuring Ehcache ACL caching

Setting up Ehcache is trivial—we'll simply declare two beans from Spring Core which manage Ehcache instantiation and expose several helpful configuration properties:

```
<bean class="org.springframework.cache.ehcache.
EhCacheManagerFactoryBean" id="ehCacheManagerBean"/>
<bean class="org.springframework.cache.ehcache.EhCacheFactoryBean"
id="ehCacheFactoryBean">
  <property name="cacheManager" ref="ehCacheManagerBean"/>
</bean>
```

Next, we'll instantiate the Ehcache ACL cache bean:

```
<bean class="org.springframework.security.acls.domain.
EhCacheBasedAclCache" id="ehCacheAclCache">
  <constructor-arg ref="ehCacheFactoryBean"/>
</bean>
```

And finally, we'll replace the reference to our `NullAclCache` implementation with the Ehcache-based one:

```
<bean class="org.springframework.security.acls.jdbc.
BasicLookupStrategy" id="lookupStrategy">
  <constructor-arg ref="dataSource"/>
  <constructor-arg ref="ehCacheAclCache"/>
  <constructor-arg ref="aclAuthzStrategy"/>
  <constructor-arg ref="aclAuditLogger"/>
</bean>
```

With these configuration changes in place (and the Ehcache runtime JARs on your classpath), ACL data will now be cached based on the Ehcache cache manager settings!

How Spring ACL uses Ehcache

While the configuration illustrated in the previous steps may seem trivial, the addition of Ehcache to your application—especially for high volume uses—carries with it the onus to perform a careful analysis, comparing the cost of caching against the cost of database queries. Understanding how Ehcache is used in Spring ACL will be critical to your planning for cache sizing and longevity.

Spring ACL will store all of the following objects (the majority of which reside in `o.s.s.acls.domain`) in cache, either as keys or values, as part of its ACL caching strategy:

- `ObjectIdentity` (implemented by `ObjectIdentityImpl`)
- `Sid` (implemented by `GrantedAuthoritySid` or `PrincipalSid`)
- `Acl` (implemented by `AclImpl`), which contains:

  `AccessControlEntry` (implemented by `AccessControlEntryImpl`)
- Your object instance's `Serializable` primary key (typically a `Long`, unless you have done significant customization to Spring ACL runtime classes)

As `BasicLookupStrategy` and `MutableAclService` are the only users of the ACL cache mechanism, use of the cache is quite straightforward. Based on the potential size of the items in the cache, it would be wise to monitor the cache during a suitable load test to assess memory usage and longevity of cache elements.

If your application is using Ehcache for other purposes (for example, Hibernate or another ORM tool), it's likely that you'll want to segregate the Ehcache instance used for ORM purposes from the instance used to store ACL data. This is typically done by supplying a unique cache name when building the `EhCacheFactoryBean` used by Spring ACL, as follows:

```
<bean class="org.springframework.cache.ehcache.EhCacheFactoryBean"
id="ehCacheFactoryBean">
  <property name="cacheManager" ref="ehCacheManagerBean"/>
  <property name="cacheName" value="springAclCacheRegion"/>
</bean>
```

Curiously, the Ehcache support in Spring Core has no formal documentation outside of the Javadoc. If your use of Spring ACL and Ehcache is significant, do note that it's likely you'll be finding yourself digging into code!

Considerations for a typical ACL deployment

Actually deploying Spring ACL in a true business application tends to be quite involved. We wrap up coverage of Spring ACL with some considerations that arise in most Spring ACL implementation scenarios.

About ACL scalability and performance modelling

For small and medium-sized applications, the addition of ACLs is quite manageable and, while it adds overhead to database storage and runtime performance, the impact is not likely to be significant. However, depending on the granularity with which ACLs and ACEs are modeled, the numbers of database rows in a medium-to large-sized application can be truly staggering, and can task even the most seasoned database administrator.

Let's assume we were to extend ACLs to cover more of the JBCP Pets application, to cover user accounts and orders, along with posts on the JBCP Pets customer-to-customer forum. We'll model the data as follows:

- All customers will have accounts
- 10% of all customers will have orders. The average number of orders per customer will be two

- Orders will be secured (read-only) per customer, but also need to be accessible (read / write) by administrators
- 10% of all customers will post on the customer-to-customer forum. The average number of posts per customer will be 20
- Posts will be secured (read-write) per customer, as well as administrators. Posts will be read-only for all other customers

Given what we know about the ACL system, we know that the database tables have the following scalability attributes:

Table	Scales with data	Scalability notes
ACL_CLASS	No	One row required per domain class.
ACL_SID	Yes (users)	One row required per role (GrantedAuthority).
		One row required for each user account (if individual domain objects are secured per user).
ACL_OBJECT_IDENTITY	Yes (domain class * instances per class)	One row required per instance of secured domain object.
ACL_ENTRY	Yes (domain object instances * individual ACE entries)	One row required per ACE, may require multiple rows for a single domain object.

We can see that ACL_CLASS doesn't really have scalability concerns (most systems will have fewer than 1000 domain classes). ACL_SID will scale linearly based on the number of users in the system. This is probably not concerning, because other user-related tables will scale in this fashion as well (user account, and so on.).

The two tables of concern are ACL_OBJECT_IDENTITY and ACL_ENTRY. If we model the estimated rows required to model an order for an individual customer, we come up with the following estimates:

Table	ACL data per order	ACL data per forum post
ACL_OBJECT_IDENTITY	One row required for a single order.	One row required for a single post.
ACL_ENTRY	Three Rows—One row required for read access by the owner (the customer SID), two rows required (one for read access, one for write access) for the administrative group SID.	Four rows—one row required for read access by the customer group SID, one row required for write access by the owner, two rows required for the administrative group SID (as with Orders)

We can then take the usage assumptions from the previous page and calculate the following ACL scalability matrix:

Table / Object	Scale factor	Estimates (Low)	Estimates (High)
Users		10,000	1,000,000
Orders	# Users * 0.1 * 2	2,000	200,000
ForumPosts	# Users * 0.1 * 20	20,000	2,000,000
ACL_SID	# Users	10,000	1,000,000
ACL_OBJECT_IDENTITY	# Orders + # Posts	22,000	2,200,000
ACL_ENTRY	(# Orders * 3) + (# Posts * 4)	86,000	8,600,000

From these projections based on only a subset of the business objects likely to be involved and secured in a typical ACL implementation, you can see that the number of database rows devoted to storing ACL information is likely to grow linearly (or faster) in relation to your actual business data. Especially in large system planning, forecasting the amount of ACL data that you are likely to use is extremely important. It is not uncommon for very complex systems to have hundreds of millions of rows related to ACL storage.

Do not discount custom development costs

Utilizing a Spring ACL-secured environment often requires significant development work above and beyond the configuration steps we've described to this point. Our sample configuration scenario has the following limitations, which would significantly impact the functionality we were to roll out ACL security to the whole site:

- SIDs and access credentials are hard-coded at application bootstrap
- No facility is provided for responding to the manipulation (create and delete) of new domain object instances, or the management of users or groups
- The application does not effectively use ACL hierarchies

When planning Spring ACL rollout across an application, you must carefully review all places where domain data is manipulated, and ensure these locations correctly update ACL and ACE rules, and invalidate caches. Typically, the securing of methods and data takes place at the service or business application layer, and the hooks required to maintain ACLs and ACEs occur at the data access layer:
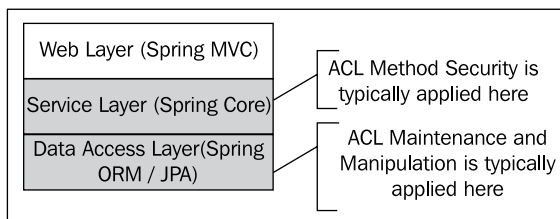
If you are dealing with a reasonably standard application architecture, with proper isolation and encapsulation of functionality, it's likely that there's an easily identified central location for these changes. On the other hand, if you're dealing with an architecture that has devolved (or was never designed well in the first place), adding ACL functionality and supporting hooks in data manipulation code can prove to be very difficult.

As previously hinted at, it's important to keep in mind that the Spring ACL architecture hasn't changed significantly since the days of Acegi 1.x, almost three years ago. In that time, many users have attempted to implement it, and have logged and documented several important restrictions, many of which are captured in the Spring Security JIRA repository (`http://jira.springframework.org/`). Issue `SEC-479` functions as a useful entry point for some of the key limitations, many of which remain unaddressed with Spring Security 3, and (if they are applicable to your situation) can require significant custom coding to work around.

Following are some of the most important, and commonly encountered, issues:

- The ACL infrastructure requires a numeric primary key. For applications that use a GUID or UUID primary key (more and more common recently due to more efficient support in modern databases), this can be a significant limitation.

- As of this writing, the JIRA issue `SEC-1140` documents the issue that the default ACL implementation does not correctly compare `Permission` bitmasks using bitwise operators. We covered this earlier in the section on permissions.

- Several inconsistencies exist between the method of configuring Spring ACL and the rest of Spring Security. In general, it is likely that you will run into areas where class delegates or properties are not exposed through DI, necessitating an override and rewrite strategy that can be time-consuming and expensive to maintain.

- The Permission bitmask is implemented as an integer, and thus has 32 possible bits. It's somewhat common to expand the default bit assignments to indicate permissions on individual object properties (for example, assigning a bit for read of social security number of an employee). Complex deployments may have well over 32 properties per domain object, in which case the only alternative would be to remodel your domain objects around this limitation.

Depending on your specific application's requirements, it is likely that you will encounter additional issues, especially with regards to the number of classes requiring change when implementing certain types of customizations.

Should I use Spring Security ACL?

Just as the details of applying Spring Security as a whole tend to be highly business dependent, so too is the application of Spring ACL support—in fact, this tends to be even more true of ACL support due to its tight coupling to business methods and domain objects. We hope that this guide to Spring ACL explained the important high- and low-level configurations and concepts required to analyze Spring ACL for use in your application, and can assist you in determining and matching its capabilities to real-world use.

Summary

In this chapter, we focused on access control list based security and the specific details of how this type of security is implemented by the Spring ACL module. We did the following:

- Reviewed the basic concept of access control lists, and the many reasons why they can be very effective solutions to authorization.

- Learned key concepts related to the Spring ACL implementation, including access control entries, SIDs, and object identity.

- Examined the database schema and logical design required to support a hierarchical ACL system.

- Configured all the required Spring Beans to enable the Spring ACL module, and enhanced one of the service interfaces to use annotated method authorization. We then tied the existing users in our database, and business objects used by the site itself, into a sample set of ACE declarations and supporting data.

- Reviewed the concepts around Spring ACL Permission handling.

- Expanded our knowledge of the Spring Security JSP tag library and SpEL expression language (for method security) to utilize ACL checks.

- Discussed the mutable ACL concept, and reviewed the basic configuration and custom coding required in a mutable ACL environment.

- Developed a custom ACL Permission, and configured the application to demonstrate its effectiveness.

- Configured and analyzed the use of the Ehcache cache manager to reduce the database impact of Spring ACL.

- Analyzed the impact and design considerations of using the Spring ACL system in a complex business application.

This wraps up the portion of the book covering core Spring Security concepts. In the following chapters, we'll dig into integrating Spring Security authentication with many types of external systems. If you don't know how the technology behind these systems (OpenID, LDAP, and so on.) works, we'll lead you through that too, so please read on!

8
Opening up to OpenID

OpenID is a very popular form of trusted identity management that allows users to manage their identity through a single trusted provider. This convenient feature provides users with the security of storing their password and personal information with the trusted OpenID provider, optionally disclosing this personal information upon request. Additionally, the OpenID-enabled website can have confidence that the users providing OpenID credentials is who they say they are.

In this chapter, we'll:

- Learn to set up your own OpenID in less than five minutes
- Configure the JBCP Pets website with a very rapid implementation of OpenID
- Learn the conceptual architecture of OpenID and how it provides your site with trustworthy user access
- Implement OpenID-based user registration
- Experiment with OpenID attribute exchange for user profile functionality
- Examine the security offered by OpenID-based login

The promising world of OpenID

The promise of OpenID as a technology is to allow users on the web to centralize their personal data and information with a trusted provider, and then use the trusted provider as a delegate to establish trustworthiness with other sites with whom the user wants to interact.

In concept, this type of login through a trusted third party has been in existence for a long time, in many different forms (Microsoft Passport, for example, became one of the more notable central login services on the web for some time). OpenID's distinct advantage is that the **OpenID Provider** needs to implement only the public OpenID protocol to be compatible with any site seeking to integrate login with OpenID. The OpenID specification itself is an open specification, which leads to the fact that there is currently a diverse population of public providers up and running the same protocol. This is an excellent recipe for healthy competition and it is good for consumer choice.

The following diagram illustrates the high-level relationship between a site integrating OpenID during the login process and OpenID providers.
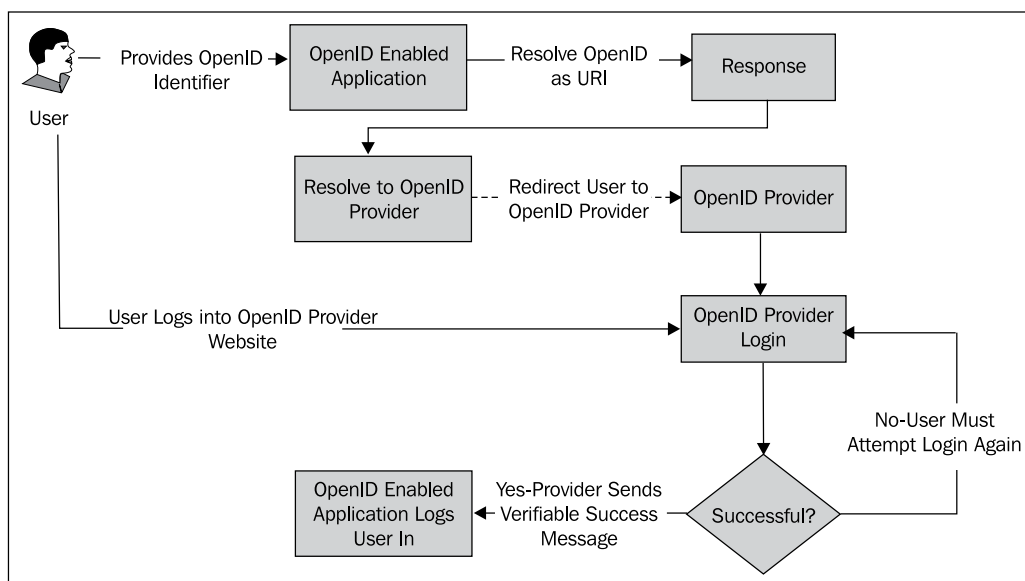
We can see that the user presents his credentials in the form of a unique named identifier, typically a **Uniform Resource Identifier** (**URI**), which is assigned to the user by their OpenID provider, and is used to uniquely identify both the user and the OpenID provider. This is commonly done by either prepending a subdomain to the URI of the OpenID provider (for example, `https://jamesgosling.myopenid.com/`), or appending a unique identifier to the URI of the OpenID provider URI (for example, `https://me.yahoo.com/jamesgosling`). We can visually see from the presented URI that both methods clearly identify both the OpenID provider (via domain name) and the unique user identifier.

> **Don't trust OpenID unequivocally!**
>
> You can see here a fundamental assumption that can fool users of the system. It is possible for us to sign up for an OpenID, which would make it appear as though we were James Gosling, even though we obviously are not. Do not make the false assumption that just because a user has a convincing-sounding OpenID (or OpenID delegate provider) they are the authentic person, without requiring additional forms of identification. Thinking about it another way, if someone came to your door just claiming he was James Gosling, would you let him in without verifying his ID?

The OpenID-enabled application then redirects the user to the OpenID provider, at which the user presents his credentials to the provider, which is then responsible for making an access decision. Once the access decision has been made by the provider, the provider redirects the user to the originating site, which is now assured of the user's authenticity.

OpenID is much easier to understand once you have tried it. Let's add OpenID to the JBCP Pets login screen now!

Signing up for an OpenID

In order to get the full value of exercises in this section (and to be able to test login), you'll need your own OpenID from one of the many available providers, of which a partial listing is available at `http://openid.net/get-an-openid/`. Common OpenID providers with which you probably already have an account are Yahoo!, AOL, Flickr, or MySpace. Google's OpenID support is slightly different, as we'll see later in this chapter when we add **Sign In with Google** support to our login page. To get full value out of the exercises in this chapter, we recommend you have accounts with at least:

- myOpenID
- Google

Enabling OpenID authentication with Spring Security

We'll see a common theme with the external authentication providers examined over the next several chapters. Spring Security provides convenient wrappers around provider integrations that are actually developed outside the Spring ecosystem.

In this vein, the openid4java project (`http://code.google.com/p/openid4java/`) provides the underlying OpenID provider discovery and request/response negotiation for the Spring Security OpenID functionality.

Writing an OpenID login form

It's typically the case that a site will present both standard (username and password) and OpenID login options on a single login page, allowing the user to select from one or the other option, as we can see in the JBCP Pets target login page.

The code for the OpenID-based form is as follows:

```
<h1>Or, Log Into Your Account with OpenID</h1>
<p>
  Please use the form below to log into your account with OpenID.
</p>
<form action="j_spring_openid_security_check" method="post">
  <label for="openid_identifier">Login</label>:
  <input id="openid_identifier" name="openid_identifier" size="20"
```

```
    maxlength="100" type="text"/>
      <img src="images/openid.png" alt="OpenID"/>
      <br />
      <input type="submit" value="Login"/>
    </form>
```

The name of the form field, `openid_identifier`, is not a coincidence. The OpenID specification recommends that implementing websites use this name for their OpenID login field, so that user agents (browsers) have the semantic knowledge of the function of this field. There are even browser plug-ins such as Verisign's OpenID SeatBelt (`https://pip.verisignlabs.com/seatbelt.do`), which take advantage of this knowledge to pre-populate your OpenID credentials into any recognizable OpenID field on a page.

You'll note that we don't offer the remember me option with OpenID login. This is due to the fact that the redirection to and from the vendor causes the **remember me** checkbox value to be lost, such that when the user's successfully authenticated, they no longer have the remember me option indicated. This is unfortunate, but ultimately increases the security of OpenID as a login mechanism for our site, as OpenID forces the user to establish a trust relationship through the provider with each and every login.

Configuring OpenID support in Spring Security

Turning on basic OpenID support, via the inclusion of a servlet filter and authentication provider, is as simple as adding a directive to our `<http>` configuration element in `dogstore-security.xml` as follows:

```
    <http auto-config="true" ...>
    <!-- Omitting content... -->
      <openid-login/>
    </http>
```

After adding this configuration element and restarting the application, you will be able to use the OpenID login form to present an OpenID and navigate through the OpenID authentication process. When you are returned to JBCP Pets, however, you will be denied access. This is because your credentials won't have any roles assigned to them. We'll take care of this next.

Adding OpenID users

As we do not yet have OpenID-enabled new user registration, we'll need to manually insert the user account (that we'll be testing) into the database, by adding them to `test-users-groups-data.sql` in our database bootstrap code. We recommend that you use myOpenID for this step (notably, you will have trouble with Yahoo!, for reasons we'll explain in a moment). If we assume that our OpenID is `https://jamesgosling.myopenid.com/`, then the SQL that we'd insert in this file is as follows:

```
insert into users(username, password, enabled, salt) values ('https://
jamesgosling.myopenid.com/','unused',true,CAST(RAND()*1000000000 AS
varchar));
insert into group_members(group_id, username) select id,'https://
jamesgosling.myopenid.com/' from groups where group_
name='Administrators';
```

You'll note that this is similar to the other data that we inserted for our traditional username-and password-based `admin` account, with the exception that we have the value `unused` for the password. We do this, of course, because OpenID-based login doesn't require that our site should store a password on behalf of the user! The observant reader will note, however, that this does not allow a user to create an arbitrary username and password, and associate it with an OpenID—we describe this process briefly later in this chapter, and you are welcome to explore how to do this as an advanced application of this technology.

At this point, you should be able to complete a full login using OpenID. The sequence of redirects is illustrated with arrows in the following screenshot:
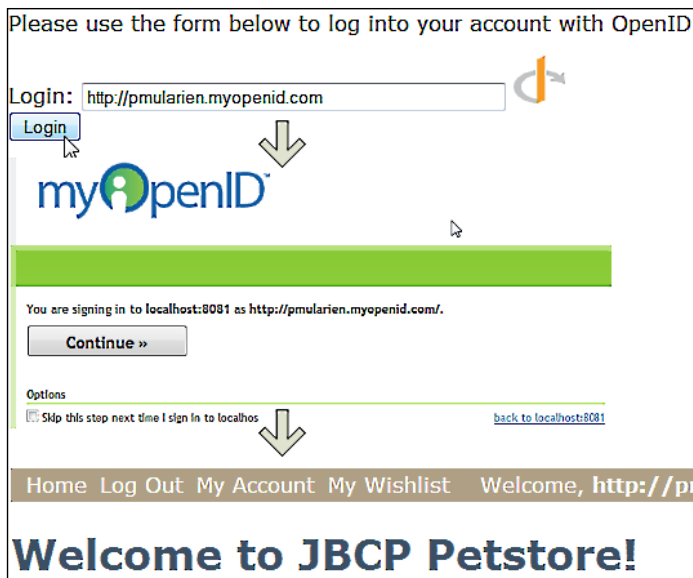
We've now OpenID-enabled JBCP Pets login! Feel free to test using several OpenID providers. You'll notice that, although the overall functionality is the same, the experience that the provider offers when reviewing and accepting the OpenID request differs greatly from provider to provider.

The OpenID user registration problem

Try using the same technique that we worked through previously with a Yahoo! OpenID—for example, `https://me.yahoo.com/pmularien`. You will find that it doesn't work, as it did with the other OpenID providers. This illustrates a key problem with the structure of OpenID, and highlights the importance of OpenID-enabled user registration.

How OpenID identifiers are resolved

The actual OpenID that Yahoo! returns will be similar to the following: `https://me.yahoo.com/pmularien#9a466`. In OpenID terminology, the identifier that the user enters in the login box is known as the **user-supplied identifier**. This identifier may not actually correspond to the identifier that uniquely identifies the user (the user's **claimed identifier**), but as part of the verification of ownership, the OpenID Provider will take care of translating the user input to the identifier that the provider can actually prove that the user owns.

> The OpenID discovery protocol and the OpenID provider itself actually have to be smart about figuring out what the user meant based on what they supply upon OpenID authentication. For example, try entering the name of an OpenID provider (for example, `www.yahoo.com`) in the OpenID login box—you'll get a slightly different interface that allows you to pick your OpenID, as you didn't supply a unique OpenID in the login box. Pretty clever! For details on this and other aspects of the OpenID specifications, check out the specifications page (on the developers page) of the OpenID Foundation website at `http://openid.net/developers/`.

Once the user is able to provide proof of ownership of their claimed identifier, the OpenID provider will return to the requesting application a normalized version of the claimed identifier, known as the **OpenID Provider Local Identifier** (or **OP-Local Identifier**). This is the final, unique identifier that the OpenID provider indicates that the user owns, and the one which will always be returned from authentication requests to the provider. Hence, this is the identifier that the JBCP Pets should be storing for user identification.

The flow of an OpenID login request handled by Spring Security proceeds
as follows:
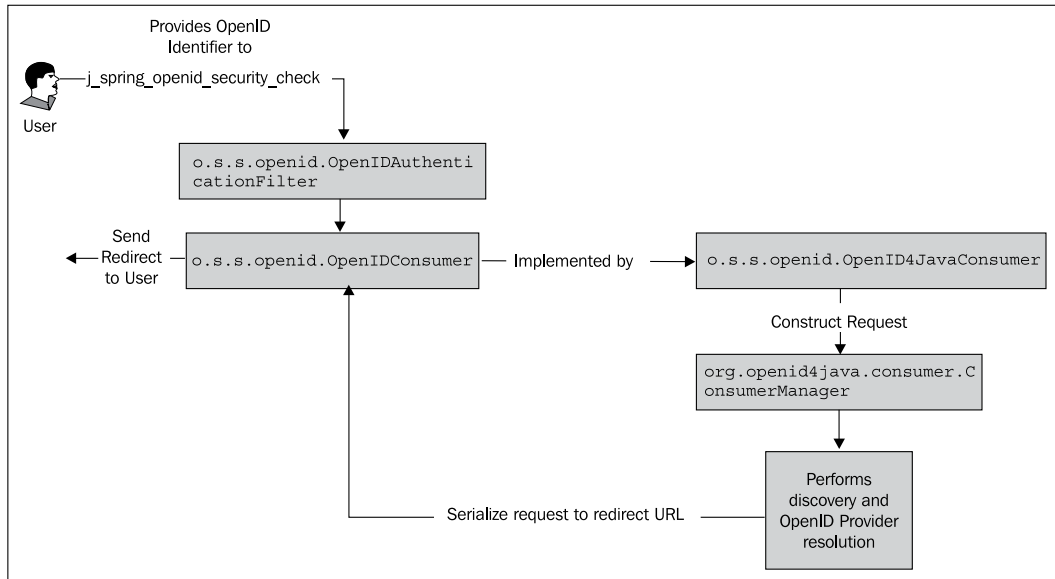
- The `o.s.s.openid.OpenIDAuthenticationFilter` is responsible
  for listening for the pseudo-URL `/j_spring_openid_security`
  `check` and responding to the user's login request, much as the
  `UsernamePasswordAuthenticationFilter` does for the `/j_spring_`
  `security_check` URL. We can see from the diagram that the `o.s.s.openid.`
  `OpenID4JavaConsumer` delegates to the openid4java library to construct
  the URL which ultimately redirects the user to the OpenID provider. The
  openid4java library (via the `org.openid4java.consumer.ConsumerManager`)
  is also responsible for the provider discovery process described earlier.

  This filter is actually used in both phases of OpenID authentication—both
  in formulating the redirect to the OpenID provider, and the handling of the
  authentication response from the provider. The response from the OpenID
  provider is a simple GET request, with a series of well-defined fields which
  are consumed and verified by the openid4java library. While you won't be
  dealing with these fields directly, some of the important ones are as follows:

Field Name	Description
openid.op_endpoint	The OpenID Provider's endpoint URL used for verification.
openid.claimed_id	The OpenID claimed identifier provided by the user.
openid.response_nonce	The nonce calculated by the provider, used to create the signature.
openid.sig	The OpenID response signature.
openid.association	The one-time use association generated by the requestor and is used to calculate the signature, and determine the validity of the response.
openid.identifier	The OP-Local identifier.

We'll examine how some of these fields are used in verifying the validity of a response. Let's look at the actors involved in processing the vendor's OpenID response:
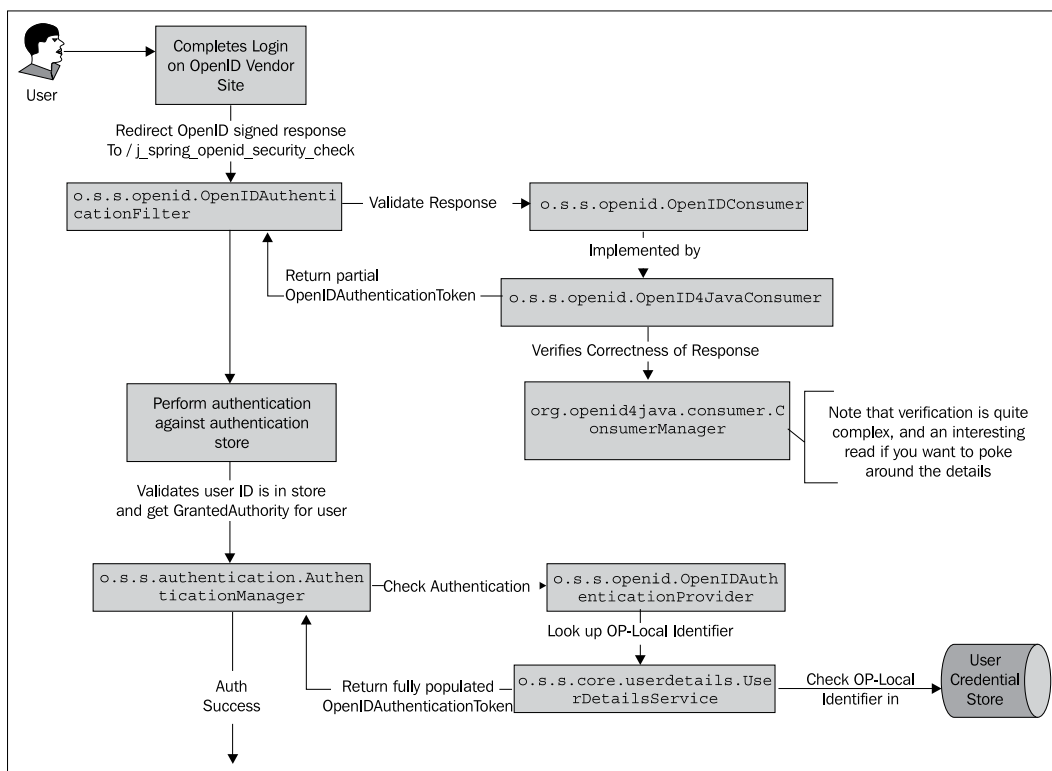
We see that the user is redirected to `/j_spring_openid_security_check` after they submit their credentials to the OpenID provider's site. The `OpenIDAuthenticationFilter` performs some rather basic checks to see if the invoking request is an OpenID request (from the JBCP Pets login form), or a possibly valid OpenID response from a provider.

Once the request is determined to be an OpenID response, a complex series of validations ensures to validate the correctness and authenticity of the response (refer to the section *Is OpenID secure?* later in this chapter for more details on this). The `OpenID4JavaConsumer` eventually returns a sparsely populated `o.s.s.openid.OpenIDAuthenticationToken`, which is used by the filter to determine whether the initial validation of the response was successful. The token is then passed along to the `AuthenticationManager`, which treats it like any other `Authentication` object.

The `o.s.s.openid.OpenIDAuthenticationProvider` ends up being responsible for performing final verification against the local authentication store (for example, `JdbcDaoImpl`). It's important to remember that what is expected in the authentication store is a username containing the OP-Local Identifier, which may not necessarily match the identifier initially supplied by the user—this is the crux of the OpenID registration problem. The flow from this point onward is very similar to traditional username/password authentication, most notably in the retrieval of appropriate group and role assignments from the `UserDetailsService`.

Implementing user registration with OpenID

For a user to be able to create an account on the JBCP Pets site, which will be OpenID enabled, they'll need to first prove that they own the identifier. Thus, we'll allow the user to supply an OpenID as part of the registration process. We've taken the liberty of adding a registration process for standard username and password authentication, which you can walk through using the **Registration** link in the header. Let's extend this registration process to allow a user to register for an account using OpenID.

Adding the OpenID registration option

Initially, we'll need to add a simple form, much like the login form (actually, exactly the same!) to the registration page, to allow the user to present and validate their OpenID identifier. Add the following to the end of `registration.jsp`:

```
<h1>Or, Register with OpenID</h1>
<p>
  Please use the form below to register your account with OpenID.
</p>
<form action="j_spring_openid_security_check" method="post">
  <label for="openid_identifier">Login</label>:
```

```
  <input id="openid_identifier" name="openid_identifier" size="50"
maxlength="100" type="text"/>
  <img src="images/openid.png" alt="OpenID"/>
  <br />
  <input type="submit" value="Login"/>
</form>
```

This form is, in fact, exactly the same as the form on the login page. How can we differentiate between a login and a registration request?

Differentiating between a login and registration request

We chose a very simple method of differentiating between a login and registration request. If the user makes a successful OpenID authentication attempt, and they haven't already got a valid account in our database, we'll assume that it's a registration request and add them to the database. There are certainly ways to refine this behavior (for example, displaying a confirmation message to the user before creating an account!), but for the purposes of our example, this is sufficient.

We'll extend the standard `AuthenticationFailureHandler` in `com.packtpub.springsecurity.security.OpenIDAuthenticationFailureHandler` class, as follows:

```
package com.packtpub.springsecurity.security;
// imports omitted
public class OpenIDAuthenticationFailureHandler extends
  SimpleUrlAuthenticationFailureHandler {
  @Override
  public void onAuthenticationFailure(HttpServletRequest request,
      HttpServletResponse response, AuthenticationException exception)
      throws IOException, ServletException {
    if(exception instanceof UsernameNotFoundException
        && exception.getAuthentication() instanceof
OpenIDAuthenticationToken
        && ((OpenIDAuthenticationToken)exception.getAuthentication()).
getStatus().equals(OpenIDAuthenticationStatus.SUCCESS)) {
      DefaultRedirectStrategy redirectStrategy = new
DefaultRedirectStrategy();
  request.getSession(true).setAttribute("USER_OPENID_CREDENTIAL", ((Us
ernameNotFoundException)exception).getExtraInformation());
      // redirect to create account page
      redirectStrategy.sendRedirect(request, response, "/
registrationOpenid.do");
```

```
        } else {
            super.onAuthenticationFailure(request, response, exception);
        }
    }
}
```

We see that this code extends a sensible superclass for default behavior, redirecting the user to the `registrationOpenid.do` URL only if the following criteria are true:

- The user has encountered a `UsernameNotFoundException`
- The user has successfully authenticated by the OpenID provider (this is validated by checking the `OpenIDAuthenticationToken's` `OpenIDAuthenticationStatus` value.

The code sets the value of the OP-Local Identifier returned by the OpenID provider in the session, so that we can retrieve it after the redirection to the OpenID registration URL.

Configuring a custom authentication failure handler

We'll need to configure this authentication failure handler with a simple adjustment in the `<openid-login>` declaration in `dogstore-security.xml`:

```
<openid-login authentication-failure-handler-ref="openIdAuthFailureHa
ndler">
The corresponding bean can be declared in dogstore-base.xml:
<bean id="openIdAuthFailureHandler" class="com.packtpub.
springsecurity.security.OpenIDAuthenticationFailureHandler">
  <property name="defaultFailureUrl" value="/login.do"/>
</bean>
```

The `defaultFailureUrl` is the location where the user will be redirected if they encounter a true login failure like providing invalid credentials.

Adding the OpenID registration functionality to the controller

The handler for OpenID-based registration is very simple, and added to the `LoginLogoutController` that we already have for standard username and password registration:

```
@RequestMapping(method=RequestMethod.GET,value="/registrationOpenid.
do")
public String registrationOpenId(HttpServletRequest request) {
```

```
    String userId = (String) request.getSession().getAttribute("USER_
 OPENID_CREDENTIAL");
    if(userId != null) {
      userService.createUser(userId, "unused", null);
      setMessage(request, "Your account has been created. Please log in
 using your OpenID.");
       return "redirect:login.do";
    } else {
      setMessage(request, "Please register using your OpenID.");
      return "redirect:registration.do";
    }
 }
```

Next, we'll modify our `IUserService` interface and `UserServiceImpl` to build
a simple `createUser` method:

```
 @Service
 public class UserServiceImpl implements IUserService {
   @Autowired
   CustomJdbcDaoImpl jdbcDao;
 // existing code omitted
   @Override
   public void createUser(String username, String password, String
 email) {
      jdbcDao.createUser(username, password, email);
   }
 }
```

You'll note that we also changed the `@Autowired` annotation to explicitly refer to our
`CustomJdbcDaoImpl`. We'll need to implement a custom `createUser` method in this
class, as follows:

```
 @Transactional
 public void createUser(String username, String password, String email)
 {
   getJdbcTemplate().update("insert into users(username, password,
 enabled, salt) values (?,?,true,CAST(RAND()*1000000000 AS varchar))",
 username, password);
   getJdbcTemplate().update("insert into group_members(group_id,
 username) select id,? from groups where group_name='Users'",
 username);
 }
```

This SQL may look familiar to you—it's the SQL we used to bootstrap users in *Chapter 4, Securing Credential Storage*. Remember that we had to create the `DatabasePasswordSecurerBean` to salt the passwords at startup? With the addition of this `createUser` method, we could actually remove the need for the bootstrap SQL, and bootstrap the users at startup using Java—how do you think we could write code to do this? Why don't you try it—it will be an effective exercise for testing your knowledge to this point.

If we did not have the custom `users` table with the `salt` field, we could simply change our `CustomJdbcDaoImpl` to inherit from `JdbcUserDetailsManager` (as discussed in Chapter 4) to pick up the `createUser` method already implemented for us:

```
public class CustomJdbcDaoImpl
    extends JdbcUserDetailsManager
    implements IChangePassword {
```

This would necessitate some minor changes to the `createUser` method of the `UserServiceImpl`:

```
@Override
public void createUser(String username, String password, String email)
{
  GrantedAuthority roleUser = new GrantedAuthorityImpl("ROLE_USER");
  UserDetails user = new User(username, password, true, true, true,
true, Arrays.asList(roleUser));
  jdbcDao.createUser(user);
}
```

You can see that there are two different ways of registering users, custom and out of the box. Each method is an effective way to handle the OpenID registration problem. Feel free to experiment with the sample application and pick the method that you like best!

Once the user has been created via our `IUserService` functionality, the user is redirected to the home page, and can successfully log in. If we wanted to enhance the user experience here, we could make some additional code changes to retain the `OpenIDAuthenticationToken` past the redirect and automatically authenticate the user.

Keep in mind that OP-Local identifiers can potentially be quite long—in fact, the OpenID 2.0 specification does not supply a maximum length for an OP-Local identifier. The default Spring Security JDBC schema provides a relatively small username column (which you may recall that we already extended from the default to 100 characters). Depending on your needs, you may wish to extend the username column further to accommodate long identifiers, or subclass some portion of the OpenID handling chain (for example, the `OpenIDAuthenticationProvider` or the `UserDetailsService`) so that identifiers which are too long are handled sensibly. This may include breaking the username into multiple columns or storing a truncated URL and a hash of the full URL, to uniquely identify the user.

Remember that authentication isn't an issue at this point, merely being able to correctly identify the user in the database, based on their OpenID. Some OpenID-enabled sites go one step further than this, and allow a level of indirection between the OpenID identifier and the username used for authentication (for example, allowing multiple OpenIDs to be associated with the same user account). The abstraction of the OpenID from the user's account name can be helpful for those users who have multiple OpenIDs from different providers that they may wish to use on your site—although this is somewhat contrary to the goals of OpenID, it does happen, and you need to keep it in mind when designing an OpenID-enabled site.

Part of the promise of OpenID, in addition to credential management and centralized trusted authentication, is the ability for users to manage their personal information in a single location and selectively release information to participating sites. This could provide, for example, a significantly richer registration experience. Let's see how Attribute Exchange hopes to solve this problem.

Attribute Exchange

One other interesting feature of OpenID is the ability for the OpenID provider to supply (upon the user's consent) typical user registration data such as name, e-mail, and date of birth, if the OpenID-enabled website requests it. This functionality is called **Attribute Exchange** (**AX**). The following diagram illustrates how a request for attribute exchange makes it into the OpenID request:
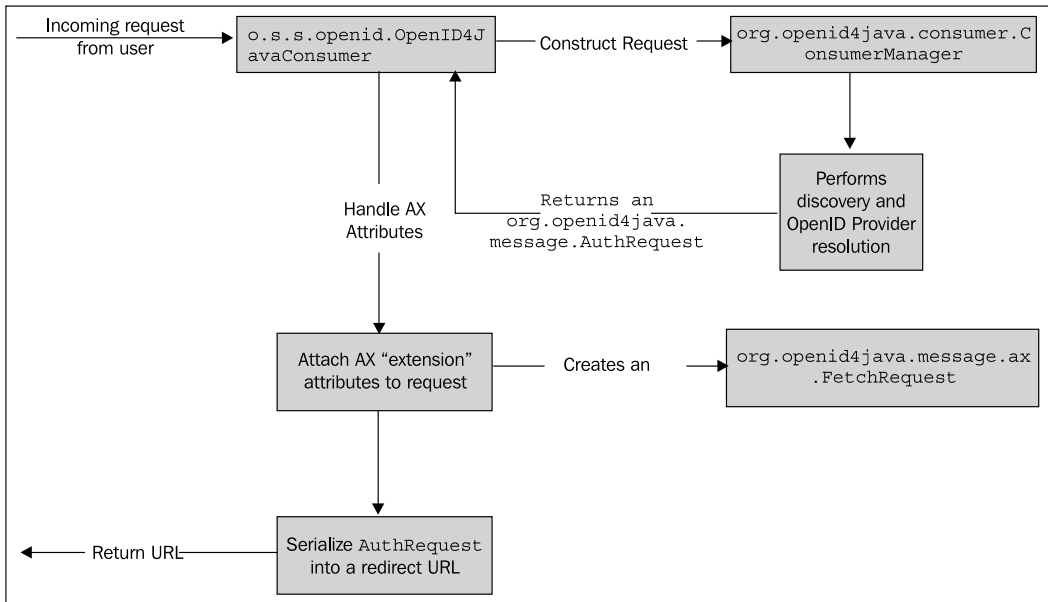
The AX attribute values (if supplied by the provider) are returned along with the rest of the OpenID response, and inserted into the `OpenIDAuthenticationToken` as a list of `o.s.s.openid.OpenIDAttribute`.

AX attributes can be arbitrarily defined by OpenID providers, but are always uniquely defined by a URI. There has been an effort to standardize the available and common attributes into a schema of sorts. Attributes such as the following are available (the full list is available at `http://www.axschema.org/types/`):

Attribute name	Description
`http://axschema.org/contact/email`	User's e-mail address
`http://axschema.org/namePerson`	User's full name

The `axschema.org` site lists over 30 different attributes, with unique URIs and descriptions. Note that you may need to reference `schema.openid.net` instead of `axschema.org` in certain cases (we'll explain why in a short time).
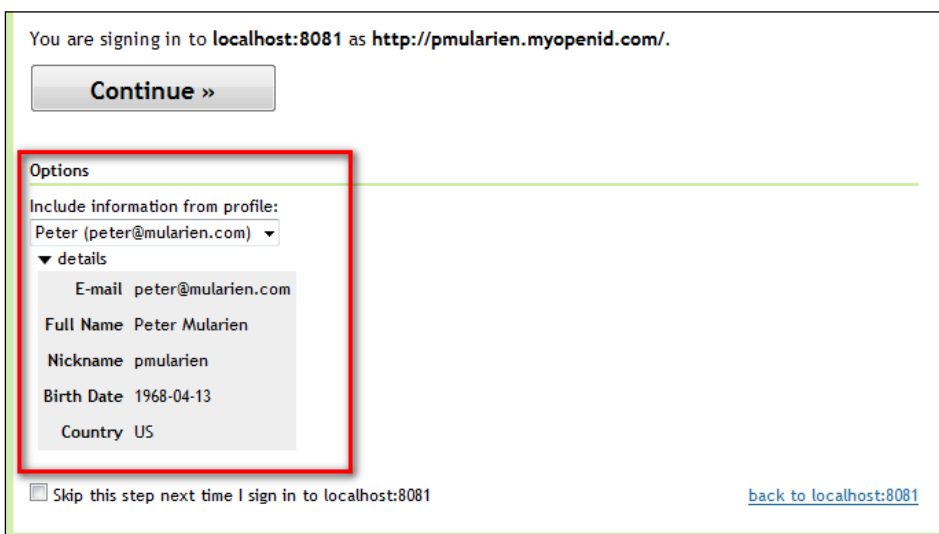
Let's see how to configure attribute exchange with Spring Security!

Enabling AX in Spring Security OpenID

Enabling AX support in Spring Security OpenID is actually quite trivial, once you know the appropriate attributes to request. We can configure AX support so that an e-mail address is requested as follows:

```
<openid-login authentication-failure-handler-ref="openIdAuthFailureHa
ndler">
  <attribute-exchange>
    <openid-attribute name="email" type="http://schema.openid.net/
contact/email" required="true"/>
  </attribute-exchange>
</openid-login>
```

For this example, we'd suggest that you log in with your myOpenID identity. You'll see that this time, when you are redirected to the provider, the provider informs you that additional information is being requested by the JBCP Pets site. In the following screenshot, we've actually included several more AX attributes in the request:

The attributes requested, if returned by the provider, are available in the `OpenIDAuthenticationToken` (returned with the successful authentication request) as name-value pairs, with the names as assigned in the `<openid-attribute>` declarations. It's up to our site to check for this data, and then do something with it. Typically, this data could be used to pre-populate a user profile or user registration form.

For investigative purposes, you can augment the `OpenIDAuthenticationFailureHandler` that we wrote to include code to print the retrieved attributes to the console:

```
request.getSession(true).setAttribute("USER_OPENID_CREDENTIAL", ((User
nameNotFoundException)exception).getExtraInformation());
OpenIDAuthenticationToken openIdAuth = (OpenIDAuthenticationToken)exce
ption.getAuthentication();
for(OpenIDAttribute attr : openIdAuth.getAttributes()) {
  System.out.printf("AX Attribute: %s, Type: %s, Count: %d\n", attr.
getName(), attr.getType(), attr.getCount());
  for(String value : attr.getValues()) {
    System.out.printf(" Value: %s\n", value);
  }
}
redirectStrategy.sendRedirect(request, response, "/registrationOpenid.
do");
```

This will produce the following output in our example:

```
AX Attribute: email, Type: http://schema.openid.net/contact/email,
Count: 1
 Value: peter@mularien.com
AX Attribute: birthDate, Type: http://schema.openid.net/birthDate,
Count: 1
 Value: 1968-04-13
AX Attribute: namePerson, Type: http://schema.openid.net/namePerson,
Count: 1
 Value: Peter Mularien
AX Attribute: nickname, Type: http://schema.openid.net/namePerson/
friendly, Count: 1
 Value: pmularien
AX Attribute: country, Type: http://schema.openid.net/contact/country/
home, Count: 1
 Value: US
```

We can see that AX data is very easily retrieved from OpenID providers, which support it and can be accessed with straightforward API calls. In typical usage scenarios, as previously discussed, AX information would be used upon registration to populate user profile or preference information, saving the user time in avoiding re-keying of information they already have in their OpenID profile.

Real-world AX support and limitations

Unfortunately, the promise of AX falls far short in reality. AX is very poorly supported by the available OpenID providers in the market, with only a handful of providers offering support (myOpenID and Google being the most prominent). Additionally, there is a lot of confusion, even among providers that do support the standard, of what attributes correspond to the data that they are willing to send. For example, to query for a user's e-mail address, the attribute name to request differs even between the two major providers who support AX!

Provider	AX attribute supported
myOpenID	`http://schema.openid.net/contact/email`
Google	`http://axschema.org/contact/email`

As of the time of this writing, there is ongoing discussion on OpenID-related mailing lists as to where and how to best document standard attributes, and to allow OpenID providers to advertise which attributes they support.

An alternative to AX, known as **Simple Registration** (**SReg**) is supported by the underlying openid4java library, but is not exposed through the Spring Security OpenID layer (by the choice of the developers). This is regrettable, because SReg is actually supported by many more providers than the ones who support AX. AX was intended as a more open and flexible alternative to SReg, but lack of adoption and standardization has hampered its uptake among providers.

Google OpenID support

Google has chosen to implement OpenID slightly differently, and doesn't assign user-friendly OpenID identifiers to users. Instead, Google expects that sites choosing to offer OpenID login through Google will use the Google canonical OpenID provider URL, and users will provide credentials directly to Google. To make this process more straightforward to users, it's common for sites to provide a **Sign in with Google** button, which invokes the Google OpenID provider. We can add this to the login page as follows:

```
<form action="j_spring_openid_security_check" method="post">
  <input name="openid_identifier" size="50" maxlength="100"
type="hidden" value="https://www.google.com/accounts/o8/id"/>
  <input type="submit" value="Sign in with Google"/>
</form>
```

We can see that the Google URL needn't be exposed to the user at all. The user is presented with the typical login screen:
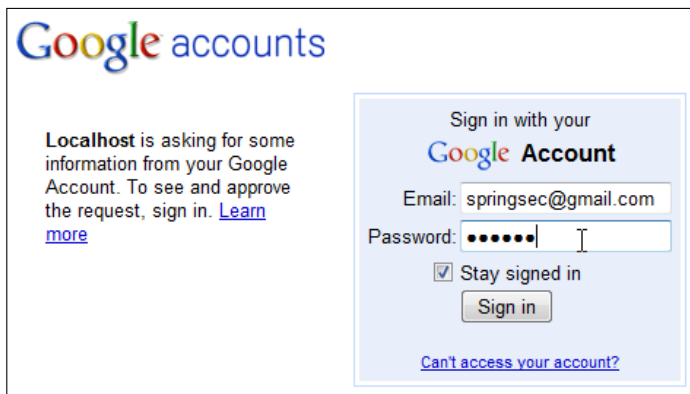
After the Google sign-in process is completed, the user's unique OP-Local Identifier will be returned, and the registration / login process can proceed as with any other OpenID provider.

Is OpenID secure?

As support for OpenID relies on the trustworthiness of the OpenID provider and the verifiability of the provider's response, security and authenticity are critical in order for the application to have confidence in the user's OpenID-based login.

Fortunately, the designers of the OpenID specification were very aware of this concern, and implemented a series of verification steps to prevent response forgery, replay attacks, and other types of tampering, which are explained in the following:

- Response forgery is prevented due to the combination of a shared secret key (created by the OpenID-enabled site prior to the initial request), and a one-way hashed message signature on the response itself. A malicious user tampering with the data in any of the response fields without having access to the shared secret key and signature algorithm would generate an invalid response.

- Replay attacks are prevented due to the inclusion of a **nonce**, or a one-time use, random key, that should be recorded by the OpenID-enabled site so that it cannot ever be reused. In this way, even a user attempting to re-issue the response URL themselves would be foiled because the receiving site would determine that their nonce had been previously used and would invalidate the request.

The most likely form of attack that could result in a compromised user interaction would be a man-in-the-middle attack, where a malicious user could intercept the user's interaction between their computer and the OpenID provider. A hypothetical attacker in this situation could be in a position to record the conversation between the user's browser and the OpenID provider, and record the secret key used when the request was initiated. The attacker in this case would need a very high level of sophistication and a reasonably complete implementation of the OpenID signature specification—in short, this is not likely to occur with any regularity.

Do note that although the openid4java library does support the use of persistent nonce tracking using JDBC, Spring Security OpenID does not currently expose this as a configuration parameter—thus nonces are tracked only in memory. This means that a replay attack could occur after a server restart, or in a clustered environment, where the in-memory store would not be replicated between JVMs on different servers.

Summary

In this chapter, we reviewed OpenID, a relatively recent technology for user authentication and credentials management. OpenID has very wide reach on the web, and has made great strides in usability and acceptance within the past year or two. Most public-facing sites on the modern web should plan on some form of OpenID support, and JBCP Pets is no exception!

In this chapter, we:

- Learned about the OpenID authentication mechanism, and explored its high level architecture and key terminology.
- Implemented OpenID Login and automatic user registration with the JBCP Pets site.
- Explored the future of OpenID profile management through the use of Attribute Exchange (AX).
- Examined the security of OpenID login responses.

We covered the most common method of web-based external authentication in this chapter. In the next chapter, we'll examine one of the most common methods of directory-based enterprise authentication: LDAP. As we move onto the next chapter, observe how external and internal authentication mechanisms differ, and how they are similar.

9
LDAP Directory Services

In this chapter, we will review the **Lightweight Directory Access Protocol** (**LDAP**) and how it might be integrated into a Spring Security enabled application to provide authentication, authorization, and user information services to interested constituents.

During the course of this chapter, we'll:

- Learn some of the basic concepts related to the LDAP protocol and server implementations
- Configure a self-contained LDAP server within Spring Security
- Enable LDAP authentication and authorization
- Understand the model behind LDAP search and user matching
- Retrieve additional user details from standard LDAP structures
- Differentiate between LDAP authentication methods, and evaluate the pros and cons of each type
- Explicitly configure Spring Security LDAP using Spring Bean declarations
- Connect to LDAP directories, including Microsoft Active Directory, for authentication

Understanding LDAP

LDAP has its roots in logical directory models dating back over thirty years—conceptually akin to a combination of an organizational chart and address book. Today, LDAP is used more and more as a way to centralize corporate user information, partitioning thousands of users into logical groups, and allowing unified sharing of user information between many disparate systems.

For security purposes, LDAP is quite commonly used to facilitate centralized username and password authentication—users' credentials are stored in the LDAP directory, and authentication requests can be made against the directory, on the user's behalf. This eases management for administrators, as user credentials—login, password, and other details—are stored in a single location in the LDAP. Additionally, organizational information such as group or team assignments, geographic location, and corporate hierarchy membership, are defined based on the user's location in the directory.

LDAP

At this point, if you have never used LDAP before, you may be wondering what it is. We'll illustrate with a screenshot of a sample LDAP schema, from the Apache Directory Server 1.5 example directory.
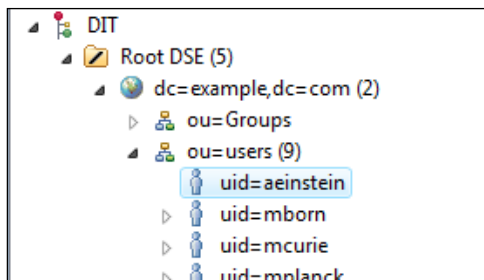
Starting at a particular user entry for Albert Einstein (highlighted in the screenshot), we can infer Mr. Einstein's organizational membership by starting at his node in the tree and moving upward. We can see that the user `aeinstein` is a member of the **organizational unit** (**ou**) `users`, which itself is a part of the domain `example.com` (the abbreviation "dc" shown in the screenshot stands for "domain component"). Previous to this are organizational elements (the DIT and Root DSE) of the LDAP tree itself, which don't concern us for now. The position of user `aeinstein` in the LDAP hierarchy is semantically and definitively meaningful—you can imagine a much more complex hierarchy easily illustrating the organizational and departmental boundaries of a huge organization.

The complete top to bottom path formed by walking the tree down to an individual leaf node forms a string composed of all intervening nodes along the way, as with Mr. Einstein's node path:

```
uid=aeinstein,ou=users,dc=example,dc=com
```

This node path is unique, and is known as a node's **distinguished name** (**DN**). The distinguished name is akin to a database primary key, allowing a node to be uniquely identified and located in a complex tree structure. We'll see a node's DN used extensively throughout the authentication and searching process with Spring Security LDAP integration.

We note that there are several other users listed at the same level of the organization as Mr. Einstein. All of these users are assumed to be within the same organizational position as Mr. Einstein. Although this example organization is relatively simple and flat, the structure of LDAP is arbitrarily flexible, with many levels of nesting and logical organization possible.

Spring Security LDAP support is assisted by the Spring LDAP module (`http://www.springsource.org/ldap`), which is actually a separate project from the core Spring Framework and Spring Security projects. It's considered to be stable, and provides a helpful set of wrappers around standard Java LDAP functionality.

Common LDAP attribute names

Each actual entry in the tree is defined by one or more **object classes**. An **object class** is a logical unit of organization, grouping a set of semantically related **attributes**. By declaring an entry in the tree as an instance of a particular object class, such as `person`, the organizer of the LDAP directory is able to provide users of the directory with a clear indication of what each element of the directory represents.

LDAP has a rich set of standard **schemas** covering available LDAP object classes and their applicable attributes (along with gobs of other information). If you are planning on doing extensive work with LDAP, it's highly advised that you review a good reference guide, such as the appendix of the *Zytrax OpenLDAP* book (`http://www.zytrax.com/books/ldap/ape/`), or the Internet2 consortium's guide to person-related schemas (`http://middleware.internet2.edu/eduperson/`).

In the last section, we were introduced to the fact that each entry in an LDAP tree has a distinguished name, which uniquely identifies it in the tree. The DN is composed of a series of attributes, one (or more) of which are used to uniquely identify the path down the tree of the entry represented by the DN. As each segment of the path described by the DN represents an LDAP attribute, you could refer to the available, well-defined LDAP schemas and object classes to determine what each of the attributes in any given DN means.

We've included some of the common attributes and their meanings in the following table. These attributes tend to be organizing attributes—meaning that they are typically used to define the organizational structure of the LDAP tree—and are ordered top to bottom in the structure that you're likely to see in a typical LDAP install.

Attribute Name	Description	Example
`dc`	Domain Component—generally the highest level of organization in an LDAP hierarchy.	`dc=jbcppets,dc=com`
`c`	Country— some LDAP hierarchies are structured at a high level by country.	`c=US`
`o`	Organization name—a parent business organization used for classifying LDAP resources.	`o=Sun Microsystems`
`ou`	Organizational unit—a divisional business organization, generally within an organization.	`ou=Product Development`
`cn`	Common name—the common name or unique or human readable name, for the object. For humans, this is usually the person's full name, while for other resources in LDAP (computers, and so on) typically the hostname.	`cn=Super Visor` `cn=Jim Bob`
`uid`	User ID—although not organizational in nature, the `uid` is generally what Spring looks for user authentication and search.	`uid=svisor`
`userPassword`	User password—stores the password for the person object to which this attribute is associated. It is typically one-way hashed using SHA or similar.	`userPassword=plaintext` `userPassword={SHA}cry ptval`

Remember that there are hundreds of standard LDAP attributes—these represent a very small fraction of those you are likely to see when integrating with a fully-populated LDAP server. The attributes in the table do, however, tend to be organizing attributes on the directory tree, and as such will probably form various search expressions or mappings that you will use to configure Spring Security to interact with the LDAP server.

Running an embedded LDAP server

Spring Security allows for the use of an embedded LDAP server for testing purposes. Much as we did with the embedded database, this allows the application to start an in-memory LDAP server and populate it with bootstrap data. Such a configuration is of course useful only for testing purposes, but it saves us a significant amount of time in configuring a standalone LDAP server.

The embedded LDAP server functionality is supported through the use of Apache Directory Server (DS) 1.5, a Java-based, open source, and fully-compliant LDAP server. You can also, in fact, use Apache DS as a standalone server as well, as it is relatively easy to configure, and is quite easy to acquire and install. The `Dependencies` directory in the sample code for this chapter includes the requisite JARs for the embedded LDAP server—if you're attempting to use it on your own, you'll have to either use Maven or download Apache DS from `http://directory.apache.org/`.

Much as the embedded HSQL database allows bootstrapping using SQL load scripts, the embedded database server provides a facility to populate the LDAP directory with an **LDAP Data Interchange Format** (**LDIF**) file. LDIF is a concise, human-and machine-readable data definition format that provides for flexible declaration of LDAP resident objects and their supporting data. Several sample LDIF files are provided with the source code for this chapter.

Configuring basic LDAP integration

Let's now enable JBCP Pets to support LDAP-based authentication. Fortunately, this is a relatively simple exercise, using the embedded LDAP server and a sample LDIF file. For this exercise, we will be using an LDIF file created for this book, intended to capture many of the common configuration scenarios with LDAP and Spring Security. We have included several more sample LDIF files, some from Apache DS 1.5, and one from the Spring Security unit tests, which you may choose to experiment with as well.

Configuring an LDAP server reference

The first step is to declare the embedded LDAP server reference in the `dogstore-security.xml` file. The LDAP server declaration occurs outside of the `<http>` element, at the same level as `<authentication-manager>`:

```
<ldap-server ldif="classpath:JBCPPets.ldif" id="ldapLocal" root="dc=jb
cppets,dc=com"/>
```

We are loading the file `JBCPPets.ldif` from the classpath, and using it to populate the LDAP server. This means (as you will remember from the embedded HSQL database bootstrap) that we will expect to see a file named `JBCPPets.ldif` in the `WEB-INF/classes` directory. The `root` attribute declares the root of the LDAP directory using the specified DN. This should correspond to the logical root DN in the LDIF file we're using.

> Be aware that for embedded LDAP servers, the `root` is required, even though the XML schema does not indicate as such. If it is not specified, or specified incorrectly, you may receive several odd errors upon initialization of the Apache DS server.
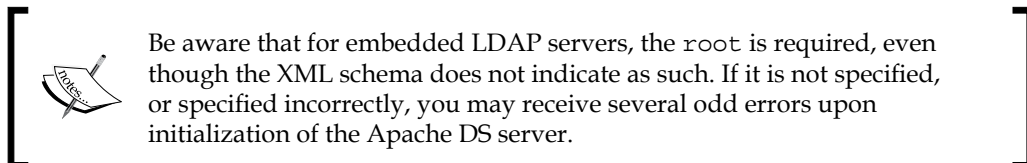
We'll reuse the bean ID defined here later in the Spring Security configuration files when we declare the LDAP user service and other configuration elements. All other attributes on the `<ldap-server>` declaration are optional for embedded LDAP mode.

Enabling the LDAP AuthenticationProvider

Next, we'll need to configure another `AuthenticationProvider` that checks user credentials against the LDAP provider. Simply add another `AuthenticationProvider` reference as follows:

```
<authentication-manager alias="authenticationManager">
<!-- Other authentication providers are here -->
  <ldap-authentication-provider server-ref="ldapLocal"
    user-search-filter="(uid={0})"
    group-search-base="ou=Groups"
  />
</authentication-manager>
```

We'll discuss these attributes a bit more later—for now, get the application back up and running and try logging in with the username `ldapguest` and the password `password`. You should be logged in!

Troubleshooting embedded LDAP

It is quite possible that you will run into hard to debug problems with embedded LDAP. Apache DS is not usually very friendly with its error messages, doubly so in Spring Security embedded mode. Some things to double-check if you can't get this simple example running are:

- Ensure you have all of the required JARs from Apache DS in your web application's classpath. There are a lot of them—it's best just to include everything (the sample code does this for you).

- Ensure the `root` attribute is set on the `<ldap-server>` declaration in your configuration file, and make sure it matches the root defined in the LDIF file that's loaded at startup. If you get errors referencing missing partitions, it's likely that either the `root` attribute was missed, or doesn't match your LDIF file.

- Be aware that a failure starting up the embedded LDAP server is not a fatal failure. In order to diagnose errors loading LDIF files, you will need to ensure that the appropriate log settings, including logging for the Apache DS server, are enabled, and at least at `ERROR` level. The LDIF loader is under the `org.apache.directory.server.protocol.shared.store` package, and this should be used to enable logging of LDIF load errors.

- If the application server shuts down non-gracefully, you may be required to delete some files in your temporary directory (`%TEMP%` on Windows systems) in order to start the server again. The error messages regarding this are (fortunately) fairly clear.

Unfortunately, embedded LDAP isn't as seamless and trivial to use as the embedded HSQL database, but it is still quite a bit easier than trying to download and configure many of the freely available external LDAP servers.

An excellent tool for troubleshooting or accessing LDAP servers in general is the Apache Directory Studio project, which offers standalone and Eclipse plugin versions. The free download is available at `http://directory.apache.org/studio/`.

Understanding how Spring LDAP authentication works

We saw that we were able to log in using a user defined in the LDIF file, and thus was present in the LDAP directory. What exactly happens when a user issues a login request for a user in LDAP? There are three basic steps to the LDAP authentication process:

- Authenticate the credentials supplied by the user against the LDAP directory
- Determine the `GrantedAuthority` that the user has, based on their information in LDAP
- Pre-load information from the LDAP entry for the user into a custom `UserDetails` object, for further use by the application

Authenticating user credentials

For the first step, authentication against the LDAP directory, a custom authentication provider is wired into the `AuthenticationManager`. The `o.s.s.ldap. authentication.LdapAuthenticationProvider` takes the user's provided credentials and verifies them against the LDAP directory, as illustrated in the following diagram:
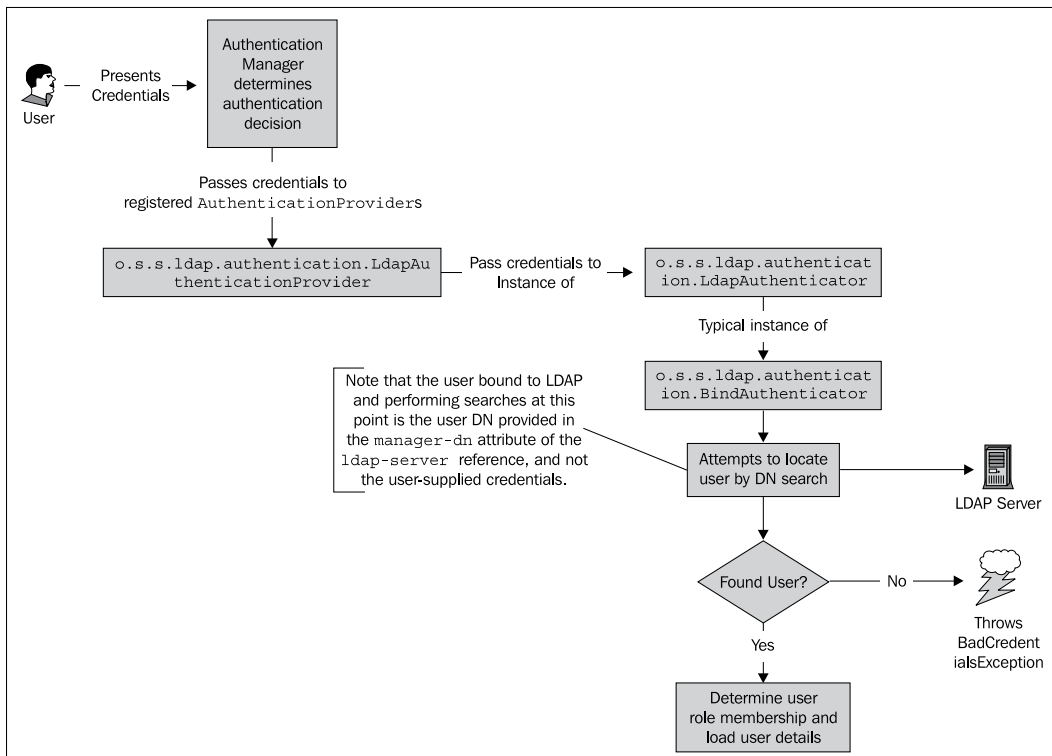
We can see that the `o.s.s.ldap.authentication.LdapAuthenticator` interface defines a delegate to allow the provider to make the authentication request in a customizable way. The implementation that we've implicitly configured to this point, `o.s.s.ldap.authentication.BindAuthenticator`, attempts to use the user's credentials to **bind** (log in) to the LDAP server as if it were the user themselves making a connection. For an embedded server, this is sufficient for our authentication needs; however, external LDAP servers may be stricter in which users are allowed to bind to the LDAP directory. Fortunately, an alternative method of authentication exists, which we will explore later in this chapter.

As noted in the diagram, keep in mind that the search is performed under an LDAP context created by the credentials specified in the `<ldap-server>` reference's `manager-dn` attribute. With an embedded server, we don't use this information, but with an external server reference, unless a `manager-dn` is supplied, anonymous binding is used. To retain some control over the public availability of information in the directory, is very common for organizations to require valid credentials to search an LDAP directory, and as such, a `manager-dn` will be almost always required in real-world scenarios. The `manager-dn` represents the full DN of a user with valid access to bind the directory and perform searches.

Determining user role membership

After the user has been successfully authenticated against the LDAP server, authorization information must be determined next. Authorization is defined by a principal's list of roles, and an LDAP authenticated user's role membership is determined as illustrated in the following diagram:
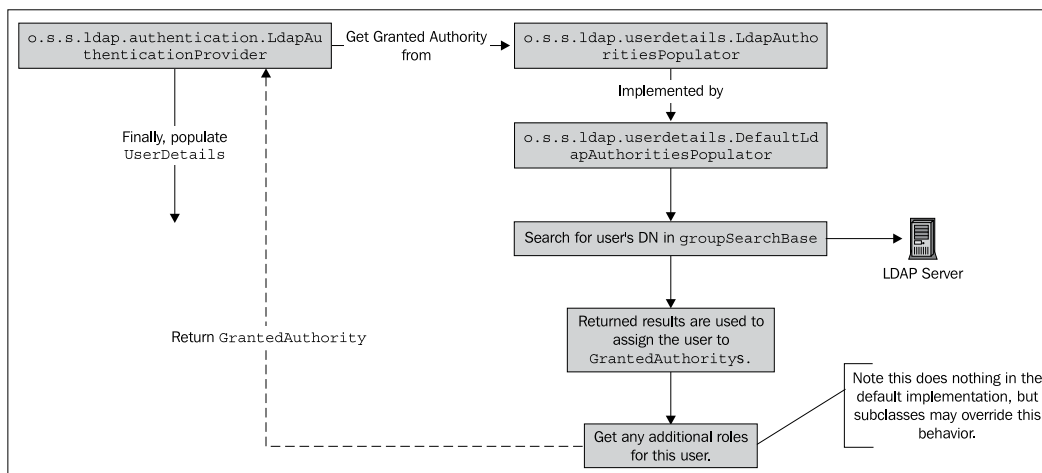
We can see that after authenticating the user against LDAP, the `LdapAuthenticationProvider` then delegates to an `LdapAuthoritiesPopulator`. The `DefaultLdapAuthoritiesPopulator` will attempt to locate the authenticated user's DN in an attribute located at or below another entry in the LDAP hierarchy.

The DN of the location searched for user role assignments is defined in the `group-search-base` attribute—in our sample, we set this to `group-search-base="ou=Groups"`. When the user's DN is located within an LDAP entry below the DN of the `group-search-base`, an attribute on the entry in which their DN is found is used to confer a role to them.

> You'll note that we mix case of these attributes—`groupSearchBase` in the class flow diagram vs. `group-search-base` referred to in the text. This is intentional—the text refers to the XML configuration attributes, while the diagram refers to the members on the relevant classes. They are similarly named, but appropriately adjusted for the context (XML and Java) in which they occur.

How Spring Security roles are associated with LDAP users can be a little confusing, so let's look at the JBCP Pets LDAP repository and see how the association of a user to a role works. The `DefaultLdapAuthoritiesPopulator` uses several attributes of the `<ldap-authentication-provider>` declaration to govern the searching of roles for the user. These attributes are used approximately in the following order.

- `group-search-base`: It defines the base DN under which the LDAP integration should look for one or more matches for the user's DN. The default value performs a search from the LDAP root, which may be expensive.

- `group-search-filter`: It defines the LDAP search filter used to match the user's DN to an attribute of an entry located under `group-search-base`. This search filter is parameterized with two parameters—the first (`{0}`) being the user's DN, and the second (`{1}`) being the user's username. The default value is (`uniqueMember={0}`).

- `group-role-attribute`: It defines the attribute of the matching entries, which will be used to compose the user's `GrantedAuthority`. The default value is `cn`.

- `role-prefix`: The prefix that will be prepended to the value found in the `group-role-attribute` to make a Spring Security `GrantedAuthority`. The default value is `ROLE_`.

This can be a little abstract and hard for new developers to follow, because it's very different than anything we've seen so far with our JDBC-based `UserDetailsService` implementations. Let's work through the login process with the `ldapguest` user in the JBCP Pets LDAP directory.

The user's DN is `uid=ldapguest,ou=Users,dc=jbcppets,dc=com` and the `group-search-base` is configured to `ou=Groups`. The LDAP tree for this `ou` is as shown in the following screenshot:
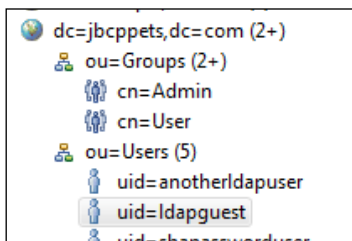
We can see that under `ou=Groups`, there are two entries (`cn=Admin` and `cn=User`). Each of these entries has the `objectClass: groupOfUniqueNames` (you'll recall we discussed object classes earlier in this chapter). This type of LDAP object allows for a multivalued list of DNs to be stored and logically grouped under the entry. The attributes of the `cn=User` are as shown in the following screenshot:

DN: cn=User,ou=Groups,dc=jbcppets,dc=com	
Attribute Description	**Value**
objectClass	groupOfUniqueNames (structural)
objectClass	top (abstract)
cn	User
uniqueMember	uid=anotherldapuser,ou=Users,dc=jbcppets,dc=com
uniqueMember	uid=ldapadmin,ou=Administrators,ou=Users,dc=jbcppets,dc=com
uniqueMember	uid=ldapguest,ou=Users,dc=jbcppets,dc=com

We can see that the `uniqueMember` attribute of the entry `cn=User` is specified for each LDAP user who is considered to be part of this group. You'll also notice that the value of the `uniqueMember` attribute is the DN of the associated user.

Working through the role search logic should be more straightforward now. Starting at `ou=Groups` (`group-search-base`), Spring will look for any `uniqueMember` attribute with the value matching the DN of the user (`group-search-filter`). When it finds entries matching these criteria, the `cn` value (`group-role-attribute`) of the entry—in this case, User, will be prepended with `ROLE_` (`role-prefix`) and then converted to upper case to form the `GrantedAuthority` for the user. It's a bit simpler to understand once we work through an example, isn't it?

> **Spring LDAP is as flexible as your gray matter**
>
> Keep in mind that although this is one way to organize an LDAP directory to be compatible with Spring Security, typical usage scenarios are exactly the opposite—an LDAP directory already exists that Spring Security needs to be wired into. In many cases you will be able to reconfigure Spring Security to deal with the hierarchy of the LDAP server; however, it's key that you plan effectively and understand how Spring is working with LDAP when it's querying. Use your brain, map out the user search and group search, and come up with the most optimal plan you can think of—keep the scope of searches as minimal and precise as possible.

If you are confused at this point, we'd suggest that you take a breather and try using Apache Directory Studio to work through browsing the embedded LDAP server configured by the running application. It can be easier to grasp the flow of Spring Security's LDAP configuration if you can attempt to search the directory yourself, by following the algorithm described previously.

Mapping additional attributes of UserDetails

Finally, once the LDAP lookup has assigned the user a set of `GrantedAuthority`s, the `o.s.s.ldap.userdetails.LdapUserDetailsMapper` will consult a `o.s.s.ldap.userdetails.UserDetailsContextMapper` to retrieve any additional details to populate the `UserDetails` for application use.

With the `<ldap-authentication-provider>` we've configured to this point, an `LdapUserDetailsMapper` will be used to populate a `UserDetails` object with information gleaned from the user's entry in the LDAP directory.
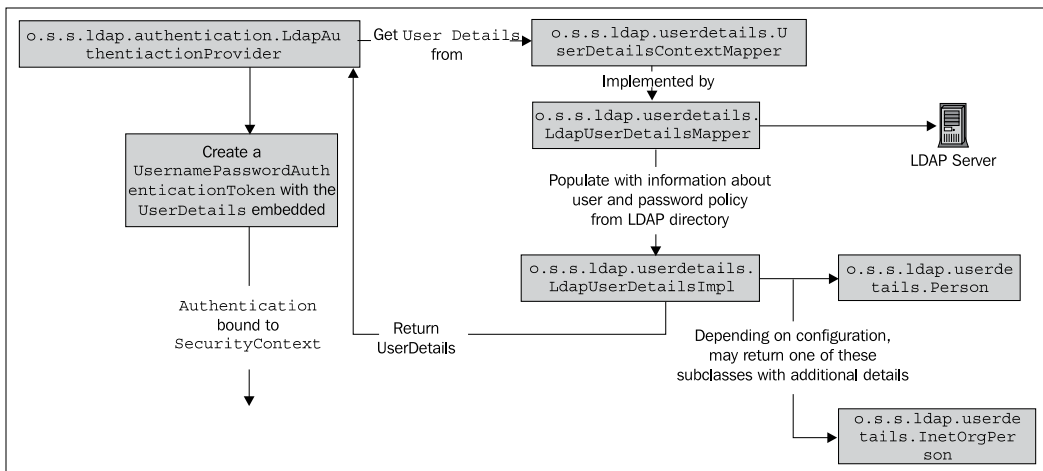
We'll see in a moment how the `UserDetailsContextMapper` can be configured to pull a wealth of information from standard LDAP `person` and `inetOrgPerson` objects. With the baseline `LdapUserDetailsMapper`, little more than username, password, and `GrantedAuthority` is stored.

Although there is more machinery involved behind the scenes in LDAP user authentication and detail retrieval, you'll note that the overall process seems somewhat similar (authenticating the user, populating `GrantedAuthoritys`) to the JDBC authentication that we've studied in prior chapters. As with JDBC authentication, there is a ability to perform advanced configuration of LDAP integration–let's dive deeper and see what's possible!

Advanced LDAP configuration

Once we get beyond the basics of LDAP integration, there's a plethora of additional configuration capability in the Spring Security LDAP module, while still remaining within the `security` XML namespace style of configuration. These include retrieval of user personal information, additional options for user authentication, and use of LDAP as a `UserDetailsService`, in conjunction with a standard `DaoAuthenticationProvider`.

Sample JBCP LDAP users

We've supplied a number of different users in the JBCP Pets LDIF file. The following quick reference chart may help you with the advanced configuration exercises, or in self-exploration. Note that the password of all users, except `userwithphone` is `password`.

Username	Role(s)	Password encoding
`ldapguest`	`ROLE_USER`	Plaintext
`anotherldapuser`	`ROLE_USER`	Plaintext
`ldapadmin`	`ROLE_USER` and `ROLE_ADMIN`	Plaintext
`shapassworduser`	`ROLE_USER`	`{sha}`
`sshapassworduser`	`ROLE_USER`	`{ssha}`
`userwithphone`	`ROLE_USER`	Plaintext (in `telephoneNumber` attribute)

We'll explain why the password encoding matters in the next section.

Password comparison versus Bind authentication

Some LDAP servers will be configured so that certain individual users are not allowed to bind directly to the server, or so that anonymous binding (what we have been using for user search to this point) is disabled. This tends to occur in very large organizations, which want a restricted set of users who are able to read information from the directory. In these cases, the standard Spring Security LDAP authentication strategy will not work, and an alternative strategy must be used, implemented by the `o.s.s.ldap.authentication.PasswordComparisonAuthenticator` (a sibling class of `BindAuthenticator`).
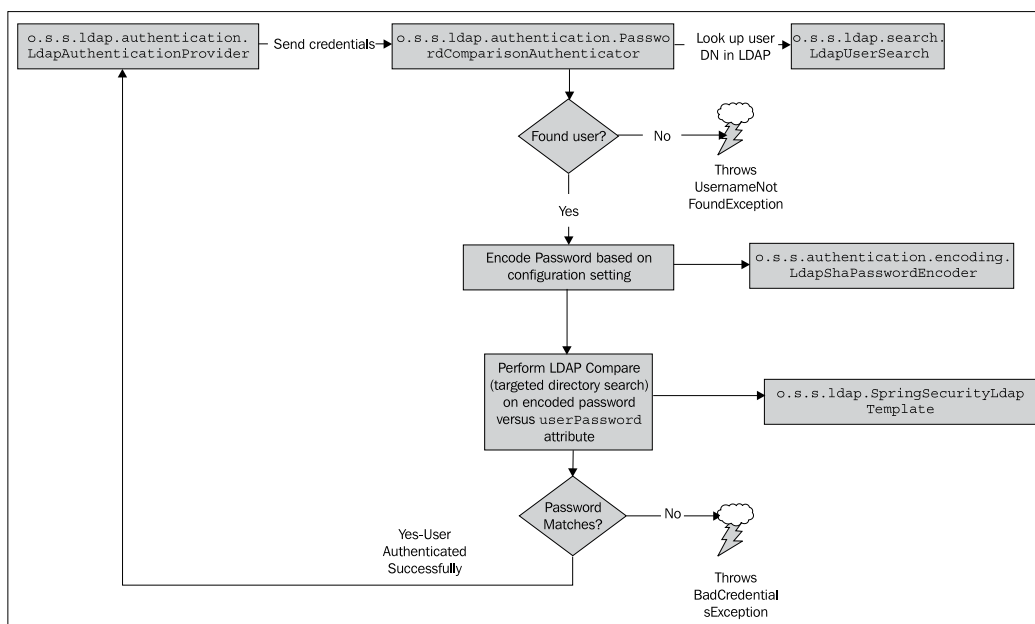
The `PasswordComparisonAuthenticator` binds to LDAP and searches for the DN matching the username provided by the user. It then compares the user supplied password with the `userPassword` attribute stored on the matching LDAP entry. If the encoded password matches, the user is authenticated, and flow proceeds as with the `BindAuthenticator`.

Configuring basic password comparison

Configuring password comparison authentication instead of bind authentication is as simple as adding a subelement to the `<ldap-authentication-provider>` declaration, as follows:

```
<ldap-authentication-provider server-ref="ldapLocal"
  user-search-filter="(uid={0})" group-search-base="ou=Groups">
  <password-compare/>
</ldap-authentication-provider>
```

The default `PasswordComparisonAuthenticator` uses the LDAP password encoding algorithm of `SHA` (recall that we discussed the SHA-1 password algorithm extensively in *Chapter 4, Securing Credential Storage*). After restarting the server, you can attempt to log in using the username `shapassworduser`, with password `password`.

LDAP password encoding and storage

LDAP has general support for a variety of password encoding algorithms, ranging from plain text to one-way hash algorithms similar to those we explored in Chapter 4, with database backed authentication. The most common storage formats for LDAP passwords are SHA (SHA-1 one-way encrypted), and SSHA (SHA-1 one-way encrypted, with a salt value). Other password formats often supported by many LDAP implementations are thoroughly documented in *RFC 2307*, *An Approach for Using LDAP as a Network Information Service* (`http://tools.ietf.org/html/rfc2307`).

The designers of RFC 2307 did a very clever thing with regards to password storage. Passwords retained in the directory are of course encoded with whatever algorithm is appropriate (SHA, and so on), but then are prefixed with the algorithm used to encode the password. This makes it very easy for the LDAP server to support multiple algorithms for password encoding. For example, an SHA encoded password is stored in the directory as:

```
{SHA}5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8
```

We can see the password storage algorithm is very clearly indicated (with the `{SHA}` notation), and stored along with the password.

SSHA is an attempt to combine the strong SHA-1 hash algorithm with password salting, to prevent dictionary attacks. As with password salting we reviewed in Chapter 4, the salt is added to the password prior to calculating the hash. When the hashed password is stored in the directory, the salt value is appended to the hashed password. The password is prepended with `{SSHA}`, so that the LDAP directory knows that the user-supplied password needs to be compared differently. The majority of modern LDAP servers utilize SSHA as the default password storage algorithm.

The drawbacks of a Password Comparison Authenticator

Now that you know a bit about how LDAP uses passwords, and we have the `PasswordComparisonAuthenticator` set up, what do you think will happen if you log in using our `sshapassworduser` user with their password stored in SSHA format? Go ahead—put the book aside and try it, and then come back.

Your login was denied, right? And yet you were still able to log in as the user with the SHA encoded password—why? The password encoding and storage didn't matter to us when we were using bind authentication—why do you think that is?

The reason it didn't matter with bind authentication was because the LDAP server was taking care of authentication and validation of the user's password. With password compare authentication, Spring Security LDAP is responsible for encoding the password in the format expected by the directory, and then matching it against the directory to validate authentication.

For security purposes, password comparison authentication can't actually read the password from the directory (reading directory passwords is often denied by security policy). Instead, the `PasswordComparisonAuthenticator` performs an LDAP search, rooted at the user's directory entry, attempting to match with a password attribute and value as determined by the password that's been encoded by Spring Security. So, when we try to log in with the `sshapassworduser`, the `PasswordComparisonAuthenticator` is encoding the password using the configured SHA algorithm, and attempting to do a simple match, which fails, as the directory password for this user is stored in SSHA format.

What happens if we change the hash algorithm for the `PasswordComparisonAuthenticator` to use SSHA as follows:

```
<password-compare hash="{ssha}"/>
```

Restart and try it out—it still doesn't work. Let's think why that might be. Remember that SSHA uses a salted password, with the salt value stored in the LDAP directory along with the password. However, `PasswordComparisonAuthenticator` is coded so that it cannot read anything from the LDAP (this typically violates security policy at companies which don't allow binding). Thus when the `PasswordComparisonAuthenticator` computes the hashed password, it has no way to determine what salt value to use.

In conclusion, the `PasswordComparisonAuthenticator` is valuable in certain limited circumstances where security of the directory itself is a concern, but will never be as flexible as straight bind authentication.

Configuring the UserDetailsContextMapper

As we noted earlier, an instance of `o.s.s.ldap.userdetails.`
`UserDetailsContextMapper` is used to map a user's entry in the LDAP to a
`UserDetails` object in memory. The default `UserDetailsContextMapper` behaves
similarly to the `JdbcDaoImpl` in the level of detail that is populated on the returned
`UserDetails`—that is to say, not a lot of information is returned aside from
username and password information.

However, an LDAP directory potentially contains many more details about
individual users than usernames, passwords, and roles. Spring Security ships with
two additional methods of pulling more user data from two of the standard LDAP
object schemas—`person` and `inetOrgPerson`.

Implicit configuration of a UserDetailsContextMapper

In order to configure a different `UserDetailsContextMapper` than the default
implementation, we simply need to declare what `LdapUserDetails` class we want
the `LdapAuthenticationProvider` to return. The security namespace parser will be
smart enough to instantiate the correct `UserDetailsContextMapper` based on the
type of `LdapUserDetails` requested.

Let's reconfigure to use the `inetOrgPerson` version of the mapper.

```
<ldap-authentication-provider server-ref="ldapLocal"
  user-search-filter="(uid={0})" group-search-base="ou=Groups"
  user-details-class="inetOrgPerson">
```

If you restart the application and attempt to log in as an LDAP user, you won't see
that anything has changed. In fact, the `UserDetailsContextMapper` has changed
behind the scenes to read additional detail in the case where attributes from the
`inetOrgPerson` schema are available in the user's directory entry.

Viewing additional user details

To assist in this area, we'll add a **View LDAP User Profile** section to the JBCP Pets
account information page. We'll use this page to illustrate how the richer `person`
and `inetOrgPerson` LDAP schemas can provide additional (optional) information
to your LDAP-enabled application.

Add the following method and logic to the `AccountController`:

```
@RequestMapping(value="/account/viewLdapUserProfile.
do",method=RequestMethod.GET)
public void showViewLdapUserProfilePage(ModelMap model) {
```

```
   final Object principal = SecurityContextHolder.getContext().
getAuthentication().getPrincipal();
   model.addAttribute("user", principal);
   if(principal instanceof LdapUserDetailsImpl) {
     model.addAttribute("isLdapUserDetails", Boolean.TRUE);
   }
   if(principal instanceof Person) {
     model.addAttribute("isLdapPerson", Boolean.TRUE);
   }
   if(principal instanceof InetOrgPerson) {
     model.addAttribute("isLdapInetOrgPerson", Boolean.TRUE);
   }
 }
}
```

This code will retrieve the `UserDetails` (principal) stored in the `Authentication` object by the `LdapAuthenticationProvider`, and determine what type of `LdapUserDetailsImpl` it is. The page code itself will then display various details depending on the type of `UserDetails` that has been bound to the user's authentication information, as we see in the following JSP code. This JSP should be placed in `WebContent/WEB-INF/views/account/viewLdapUserProfile.jsp`.

```
<!-- Common Header and Footer Omitted -->
<h1>View Profile</h1>
<p>
  Some information about you, from LDAP:
</p>
<ul>
  <li><strong>Username:</strong> ${user.username}</li>
  <li><strong>DN:</strong> ${user.dn}</li>
  <c:if test="${isLdapPerson}">
    <li><strong>Description:</strong> ${user.description}</li>
    <li><strong>Telephone:</strong> ${user.telephoneNumber}</li>
    <li><strong>Full Name(s):</strong>
    <c:forEach items="${user.cn}" var="cn">
      ${cn}<br />
    </c:forEach>
    </li>
  </c:if>
  <c:if test="${isLdapInetOrgPerson}">
    <li><strong>Email:</strong> ${user.mail}</li>
    <li><strong>Street:</strong> ${user.street}</li>
  </c:if>
</ul>
```

We can also add a link to `WebContent/WEB-INF/views/account/home.jsp` as follows:

```
<li><a href="viewLdapUserProfile.do">View LDAP User Profile</a></li>
```

We've added two more users that you can use to examine the differences in available data elements: `shapasswordperson` and `shapasswordinetorgperson`. Restart the server, and examine the **View LDAP User Profile** page for each of the three types of users (non-`person`, `person`, and `inetOrgPerson`). You'll note that, when `user-details-class` is configured to use `inetOrgPerson`, although an `o.s.s.ldap.userdetails.InetOrgPerson` is what is returned, the fields may or may not be populated depending on the available attributes in the directory entry.

In fact, `inetOrgPerson` has many more attributes than what we've illustrated on this simple page. You can review the full list in RFC 2798, *Definition of the inetOrgPerson LDAP Object Class* (`http://tools.ietf.org/html/rfc2798`).

One thing you may notice is that there is no facility to support additional attributes that may be specified on an entry, but don't fall into a standard schema. The standard `UserDetailsContextMapper`s don't support arbitrary lists of attributes, but it is possible nonetheless to customize with a reference to your own `UserDetailsContextMapper` through the use of the `user-context-mapper-ref` attribute.

Using an alternate password attribute

In some cases, it may be necessary to use an alternate LDAP attribute, instead of `userPassword`, for authentication purposes. This can happen on occasion when companies have deployed custom LDAP schemas, or don't have the requirement of strong password management (arguably, this is never a good idea, but it definitely does occur in the real world).

The `PasswordComparisonAuthenticator` also supports the ability to verify the user's password against an alternate LDAP entry attribute, instead of the standard `userPassword` attribute. This is very easy to configure, and we can demonstrate a simple example using the plaintext `telephoneNumber` attribute, as follows:

```
<ldap-authentication-provider server-ref="ldapLocal"
  user-search-filter="(uid={0})" group-search-base="ou=Groups"
  user-details-class="inetOrgPerson">
  <password-compare hash="plaintext" password-attribute="telephoneNum
ber"/>
</ldap-authentication-provider>
```

We can restart the server and attempt to log in with the user `userwithphone` and password (telephone number) `1112223333`.

Of course, this type of authentication has all the perils we discussed earlier about `PasswordComparisonAuthenticator` based authentication; however, it's good to be aware of if on the off chance that it comes up with an LDAP implementation.

Using LDAP as a UserDetailsService

One thing to note is that LDAP may also be used as a `UserDetailsService`. Remember that a `UserDetailsService` is required to enable various other bits of functionality in the Spring Security infrastructure, including the remember me and OpenID authentication features.

Configuration of LDAP as a `UserDetailsService` functions very similarly to the configuration of an LDAP `AuthenticationProvider`. Like a JDBC `UserDetailsService`, an LDAP `UserDetailsService` is configured as a sibling to the `<http>` declaration.

```
<ldap-user-service id="ldapUserService" server-ref="ldapLocal"
    user-search-filter="(uid={0})" group-search-base="ou=Groups"/>
```

Functionally, `o.s.s.ldap.userdetails.LdapUserDetailsService` is configured almost exactly the same as the `LdapAuthenticationProvider`, with the exception that there is no attempt to use the principal's username to bind to LDAP. Instead, the credentials supplied by the `<ldap-server>` reference itself are used to perform the user lookup.

> Do not make the very common mistake of configuring an `<authentication-provider>` with a `user-details-service-ref` referring to an `LdapUserDetailsService`, if you are intending to authenticate the user against LDAP itself!

As noted above, the `LdapUserDetailsService` uses the `manager-dn` supplied with the `<ldap-server>` declaration to get its information—this means that it does not attempt to bind the user to LDAP, and as such may not behave as you expect. The `LdapUserDetailsService` is typically used to back other components of the system, such as OpenID login or remember me where the authentication is already handled, but detailed information about the user is not supplied as part of the authentication process.

Notes about remember me with an LDAP UserDetailsService

You may note at this point that an attempt to start the server will fail with an error similar to the following:

```
More than one UserDetailsService registered. Please use a specific Id
reference in <remember-me/> <openid-login/> or <x509 /> elements.
```

If we think back to the introduction of the remember me functionality in *Chapter 3*, *Enhancing the User Experience* and *Chapter 4*, *Securing Credential Storage*, we recall that remember me relies on a `UserDetailsService` to look up the username identified in the remember me cookie. Unfortunately, it's possible only for `AbstractRememberMeServices` to look to a single `UserDetailsService` for user information. Thus, we can only use the remember me feature with one of the configured `UserDetailsServices` and not both. This makes it difficult to realistically combine LDAP authentication with JDBC while using the same login page, and still making the remember me feature available to users. Adjusting the `<remember-me>` declaration to refer to a single `UserDetailsService` (by Spring Bean ID) is sufficiently specific to tell Spring Security what you intended.

Configuration for an In-Memory remember me service

Another note about using remember me functionality with LDAP—you must (usually) use the JDBC-based persistent token remember me service in order to provide remember me functionality to users authenticated using LDAP.

You may remember from discussion of the composition of the remember me cookie in Chapter 3 that the in-memory remember me algorithm (in `InMemoryTokenRepositoryImpl`) relies on both the username and password being available in the `UserDetails` object. In many cases, LDAP servers are configured not to allow reading of `userPassword` attributes (this is why `PasswordComparisonAuthenticator` is written the way it is), and as such it is likely that `LdapUserDetailsMapper` will be unable to populate the password field of the `UserDetails` object. Lack of this critical data will cause creation of the remember me cookie to fail, rather than potentially compromising the security of the cookie.

Avoiding this problem is simple—configure a JDBC resident remember me cookie store, which relies only on the username to validate the cookie (as discussed in Chapter 4).

Integrating with an external LDAP server

It is likely that once you test basic integration with the embedded LDAP server you will want to interact with an external LDAP server. Fortunately, this is very straightforward, and can be done using slightly different syntax along with the same `<ldap-server>` instruction we provided to set up the embedded LDAP server.

The following code is a sample configuration used to connect to an external LDAP server on port 10389:

```
<ldap-server url="ldap://localhost:10389/dc=jbcppets,dc=com"
id="ldapLocal"
  manager-dn="uid=admin,ou=system" manager-password="secret"/>
```

The notable differences here (aside from the LDAP URL) are that the DN and password for an account are provided. The account (which is actually optional) should be allowed to bind to the directory and perform searches across all relevant DNs for users and group information. The binding resulting from the application of these credentials against the LDAP server URL is what is used for remaining LDAP operations across the LDAP secured system.

Be aware that many LDAP servers also support SSL-encrypted LDAP (LDAPS)—this is of course preferred for security purposes, and is supported by the Spring LDAP stack. Simply use `ldaps://` at the beginning of the LDAP server URL. LDAPS typically runs on TCP port 636.

Note that there are many commercial and non-commercial implementations of LDAP. The exact configuration parameters that you will use for connectivity, user binding, and population of `GrantedAuthority`s will wholly depend on both the vendor and the structure of the directory. We cover one very common LDAP implementation, that is, Microsoft Active Directory, in the next section.

Explicit LDAP bean configuration

In this section, we'll lead you through the set of bean configurations required to explicitly configure both a connection to an external LDAP server and the `LdapAuthenticationProvider` required to support authentication against an external server. As with other explicit bean-based configurations, you really want to avoid doing this, unless you find yourself in a situation where the capabilities of the `security` namespace style of configuration will not support your business or technical requirements—in which case, read on!

Configuring an external LDAP server reference

To implement this configuration, we'll assume that we have a local LDAP server running on port 10389, with the same configuration corresponding to the `<ldap-server>` example provided in the previous section. The required bean definition can be supplied in `dogstore-base.xml`, as follows:

```
<bean class="org.springframework.security.ldap.
DefaultSpringSecurityContextSource" id="ldapServer">
  <constructor-arg value="ldap://localhost:10389/dc=jbcppets,dc=com"/>
  <property name="userDn" value="uid=admin,ou=system"/>
  <property name="password" value="secret"/>
</bean>
```

Next, we'll need to configure the `LdapAuthenticationProvider`, which is a bit more complex.

Configuring an LdapAuthenticationProvider

If you've read and understood the explanations throughout this chapter describing how Spring Security LDAP authentication works behind the scenes, this bean configuration will be perfectly understandable, albeit a bit complex. We'll configure an `LdapAuthenticationProvider` with the following characteristics:

- User credential binding authentication (not password compare).
- Use of the `InetOrgPerson` in the `UserDetailsContextMapper`.

Let's get to it—we'll declare the `LdapAuthenticationProvider` first:

```
<bean class="org.springframework.security.ldap.authentication.
LdapAuthenticationProvider" id="ldapAuthProvider">
  <constructor-arg ref="ldapBindAuthenticator"/>
  <constructor-arg ref="ldapAuthoritiesPopulator"/>
  <property name="userDetailsContextMapper" ref="ldapUserDetailsConte
xtMapper"/>
</bean>
```

Next, the `BindAuthenticator` and the supporting `FilterBasedLdapUserSearch` bean (used to locate the user's DN in the LDAP directory prior to binding):

```
<bean class="org.springframework.security.ldap.authentication.
BindAuthenticator" id="ldapBindAuthenticator">
  <constructor-arg ref="ldapServer"/>
  <property name="userSearch" ref="ldapSearchBean"/>
```

```
    </bean>
    <bean class="org.springframework.security.ldap.search.
    FilterBasedLdapUserSearch" id="ldapSearchBean">
      <constructor-arg value=""/> <!-- user-search-base -->
      <constructor-arg value="(uid={0})"/> <!-- user-search-filter -->
      <constructor-arg ref="ldapServer"/>
    </bean>
```

Finally, the `LdapAuthoritiesPopulator` and `UserDetailsContextMapper`, which perform the roles we've examined earlier in the chapter:

```
    <bean class="org.springframework.security.ldap.userdetails.
    DefaultLdapAuthoritiesPopulator" id="ldapAuthoritiesPopulator">
      <constructor-arg ref="ldapServer"/>
      <constructor-arg value="ou=Groups"/>
      <property name="groupSearchFilter" value="(uniqueMember={0})"/>
    </bean>
    <bean class="org.springframework.security.ldap.userdetails.
    InetOrgPersonContextMapper" id="ldapUserDetailsContextMapper"/>
```

The last element that's required is the reference to our explicitly configured `LdapAuthenticationProvider`, which we'll declare by reference in `dogstore-security.xml`:

```
    <authentication-manager alias="authenticationManager">
      <authentication-provider ref="ldapAuthProvider"/>
    </authentication-manager>
```

At this point, we have fully configured LDAP authentication with explicit Spring Bean notation. As with the explicit bean configuration we first introduced in *Chapter 6*, *Advanced Configuration and Extension*, use of this technique in LDAP integration is useful in the few cases where the `security` namespace does not expose certain configuration attributes, or where custom implementation classes are required to provide functionality tailored to a particular business scenario. We'll explore one such scenario next when we examine how to connect to Microsoft Active Directory via LDAP.

Integrating with Microsoft Active Directory via LDAP

One of the convenient features of Microsoft Active Directory is it's seamless integration not only with Microsoft Windows-based network architectures, but it can also be configured to expose the contents of Active Directory using the LDAP protocol. If you are a Windows shop, it is probable that any LDAP integration you do will be against your Active Directory instance.

Depending on your configuration of Microsoft Active Directory (and the directory administrator's willingness to configure it to support Spring Security LDAP), you may have a difficult time not with the authentication and binding process, but with the mapping of Active Directory information to user `GrantedAuthoritys` within the Spring Security system.

The sample, Active Directory LDAP tree for JBCP Pets corporate within our LDAP browser is as shown in the following screenshot:
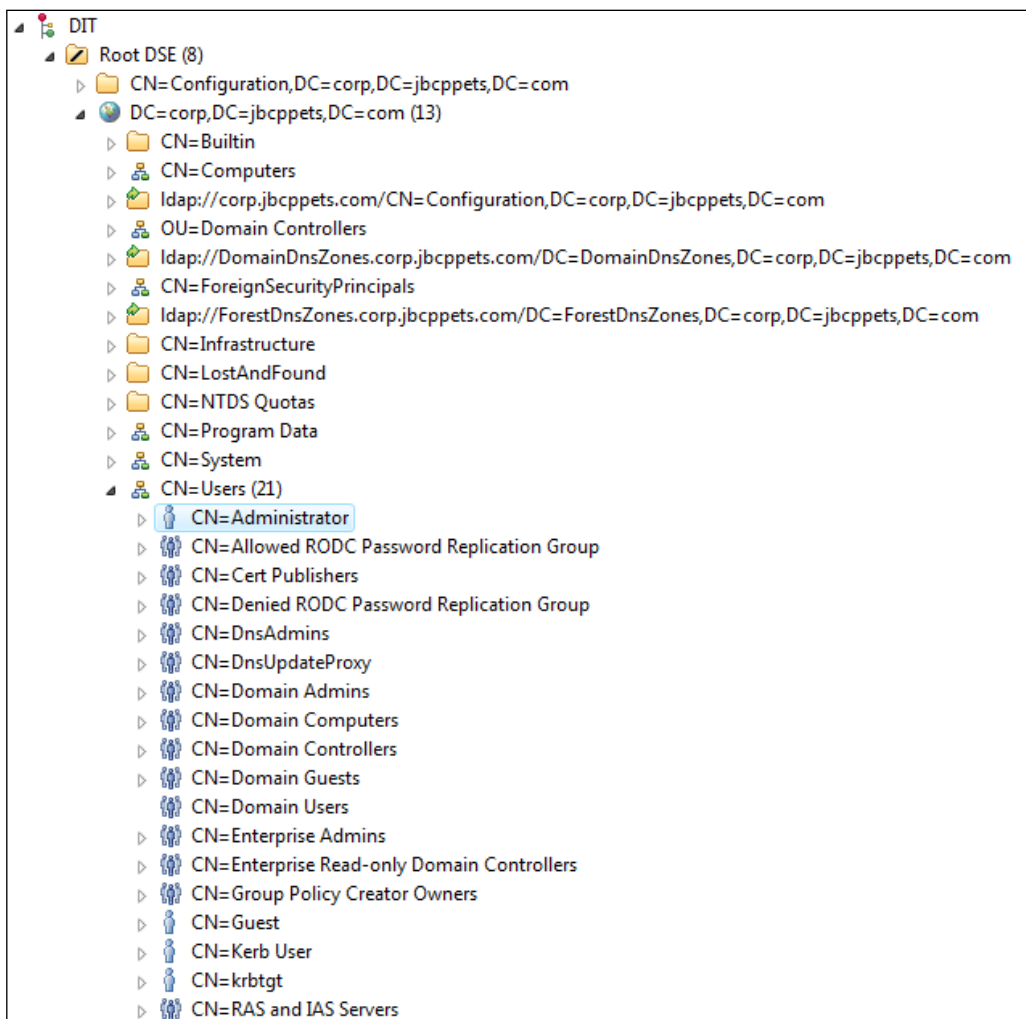
What you do not see here is an `ou=Groups` as we saw in our sample LDAP structure earlier—this is because Active Directory stores group membership as attributes on the LDAP entries of users themselves. Out of the box (as of the time of publishing), Spring Security does not offer an `LdapAuthoritiesPopulator` that can be configured to support the structure of a typical Active Directory LDAP tree.

Let's use our recently acquired knowledge of explicit bean configuration to write a simple `LdapAuthoritiesPopulator` implementation, which simply grants `ROLE_USER` to anyone who has successfully authenticated against the LDAP directory. We'll call this class as `com.packtpub.springsecurity.security.SimpleRoleGrantingLdapAuthoritiesPopulator`:

```
public class SimpleRoleGrantingLdapAuthoritiesPopulator implements
    LdapAuthoritiesPopulator {
  protected String role = "ROLE_USER";
  public Collection<GrantedAuthority> getGrantedAuthorities(
      DirContextOperations userData, String username) {
    GrantedAuthority ga = new GrantedAuthorityImpl(role);
    return Arrays.asList(ga);
  }
  public String getRole() {
    return role;
  }
  public void setRole(String role) {
    this.role = role;
  }
}
```

Not very exciting, but keep in mind that you can use the `o.s.ldap.core.DirContextOperations` object passed to perform many types of LDAP operations if required, such as querying the directory for information about the user, and using that information to fill out the roles returned.

Let's see how to alter our configuration to support our Active Directory structure now, assuming we are starting with the bean configuration detailed in the previous section:

```
    <bean class="org.springframework.security.ldap.
  DefaultSpringSecurityContextSource" id="ldapServer">
      <constructor-arg value="ldap://corp.jbcppets.com/dc=corp,dc=jbcpp
  ets,dc=com"/>
      <property name="userDn" value="CN=Administrator,CN=Users,DC=corp,D
  C=jbcppets,DC=com"/>
      <property name="password" value="admin123!"/>
    </bean>
    <bean class="org.springframework.security.ldap.search.
  FilterBasedLdapUserSearch" id="ldapSearchBean">
      <constructor-arg value="CN=Users"/>
```

```
    <constructor-arg value="(sAMAccountName={0})"/>
    <constructor-arg ref="ldapServer"/>
  </bean>
  <bean class="com.packtpub.springsecurity.security.
SimpleRoleGrantingLdapAuthoritiesPopulator" id="ldapAuthoritiesPopula
tor"/>
```

The attribute `sAMAccountName` is the Active Directory equivalent of the `uid` attribute we used in a standard LDAP entry—the remainder of the configuration changes are typical of a standard Active Directory configuration.

Although most Active Directory LDAP integrations are likely to be more complex than this example, this should give you a starting point to jump off and explore from with your conceptual understanding of the inner workings of Spring Security LDAP integration, supporting even a complex integration will be much easier.

Also be aware that another type of integration with Active Directory exists—Kerberos authentication, which is a part of the Spring Security Extensions project, and covered in *Chapter 12*, *Spring Security Extensions*. You may wish to review both types of integration, and determine which is more suitable for your needs (we will compare both LDAP and Kerberos authentication for Active Directory in Chapter 12).

Delegating role discovery to a UserDetailsService

One final technique for populating user roles that is available to use with explicit bean configuration is support for looking up a user by username in a `UserDetailsService`, and getting their `GrantedAuthority` from this source. Configuration is as simple as declaring the bean, with a reference to a `UserDetailsService` (such as a JDBC-based one or such as the one we have been using in prior chapters):

```
  <bean class="org.springframework.security.ldap.authentication.
UserDetailsServiceLdapAuthoritiesPopulator" id="ldapAuthoritiesPopula
tor">
    <constructor-arg ref="jdbcUserServiceCustom"/>
  </bean>
```

The logistical and managerial problem you may foresee with this is that the usernames and roles must be managed both in the LDAP and the repository used by the `UserDetailsService`—this is probably not a scalable model with a large user base.

The more common use of this scenario is when LDAP authentication is required to ensure that users of the secured application are valid corporate users, but the application itself wants to store authorization information. This keeps potentially application-specific data out of the LDAP directory, which can be a beneficial separation of concerns.

Summary

We have seen that LDAP servers can be relied upon to provide authentication and authorization information, as well as rich user profile information, when requested. In this chapter, we covered:

- LDAP terminology and concepts, and understand how LDAP directories might be commonly organized to work with Spring Security

- Configuration of both standalone (embedded) and external LDAP servers from a Spring Security configuration file

- Authentication and authorization of users against LDAP repositories, and subsequent mapping to Spring Security actors

- Differences in authentication schemes and password storage and security mechanisms in LDAP—and how they are treated in Spring Security

- Mapping user detail attributes from the LDAP directory to the `UserDetails` object for rich information exchange between LDAP and the Spring-enabled application

- Explicit bean configuration for LDAP, and the pros and cons of this approach

10
Single Sign On with Central Authentication Service

In this chapter, we'll examine the use of **Central Authentication Service** (**CAS**) as a single sign-on portal for Spring Security-based applications:

During the course of this chapter, we'll:

- Learn about CAS, its architecture, and benefits to system administrators and organizations of any size
- Understand how Spring Security can be reconfigured to handle interception of authentication requests and redirect to CAS
- Configure the JBCP Pets application to utilize CAS single sign-on
- Integrate CAS with LDAP, and pass data from LDAP to Spring Security via CAS
- Review the differences between the CAS 2.0 protocol and SAML 1.1

Introducing Central Authentication Service

Central Authentication Service is an open source single sign-on portal, providing centralized access control, and authentication to web-based resources within an organization. The benefits of CAS are numerous to administrators supporting many applications and diverse user communities:

- Individual or group access to resources (applications) can be configured in one location

- Broad support for a wide variety of authentication stores (to centralize user management) provides a single point of authentication and control to a widespread, cross-machine environment

- Wide authentication support is provided for web-based and non-web based Java applications through CAS client libraries

- A single point of reference for user credentials (via CAS), so that CAS client applications are not required to have any knowledge of the user's credentials, or how to verify them

We'll focus in this chapter not as much on the management of CAS, but specifically on authentication and how CAS can act as an authentication point for users of our site. Although CAS is commonly seen in intranet environments for enterprise or educational institutions, it can also be found in use at high-profile locations such as Sony Online Entertainment's and Dun and Bradstreet's public-facing sites.

High level CAS authentication flow

The basic authentication flow of CAS proceeds via the following actions:

- The user attempts to access a protected resource on the website.

- The user is redirected by the website security mechanism to the CAS portal to present credentials.

- The CAS portal is responsible for user authentication. If successfully authenticated to CAS, the user is redirected back to the protected resource with a unique CAS **ticket** assigned to the request.

- The website security mechanism calls back to the CAS server to validate that the ticket is acceptable (is valid, has not expired, and so on.). The CAS server responds with an **assertion** indicating that trust has been established. If the ticket is acceptable, trust has been established, and the user may proceed via normal authorization checking.

Visually, this behaves as illustrated in the following diagram:
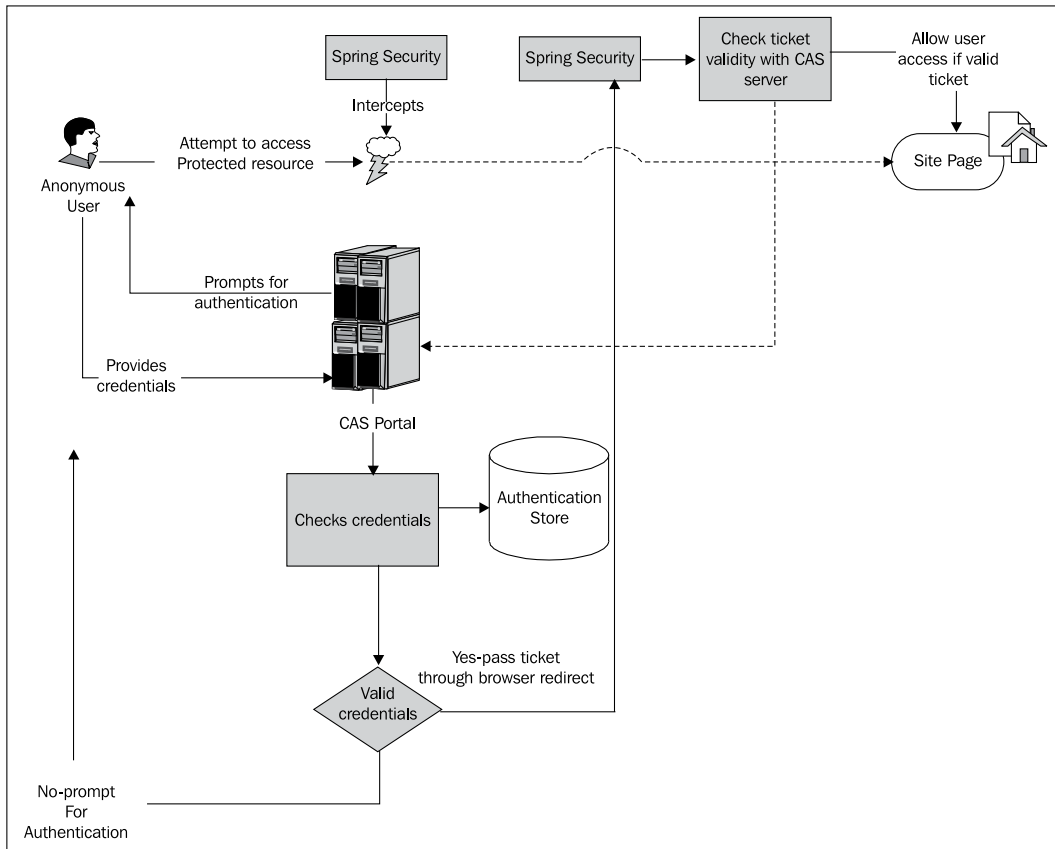
We can see that there is a high level of interaction between the CAS server and the secured application, with several data exchange handshakes required before trust of the user can be established. The result of this complexity is a single sign-on protocol that is quite hard to spoof through common techniques (assuming other network security precautions, such as use of SSL and network monitoring are in place).

Now that we understand how CAS authentication works in general, let's see how it applies to Spring Security.

Spring Security and CAS

Spring Security has a strong integration capability with CAS, although not as tightly integrated into the `security` namespace style of configuration as the OpenID and LDAP integrations we've explored thus far in the latter portion of this book. Instead, much of the configuration relies on bean wiring and configuration-by-reference from the `security` namespace elements to bean declarations.

The two basic pieces of CAS authentication involve the following:

- Replacement of the standard `AuthenticationEntryPoint`—which typically handles redirection of unauthenticated users to the login page—with an implementation that redirects the user instead to the CAS portal.

- Handling of the ticket assignment and presentment when the user is redirected back from the CAS portal to the protected resource, through the use of a custom servlet filter.

An important thing to understand about CAS is that in typical deployments, CAS is intended to replace all alternative login mechanisms of your application. As such, once you configure CAS for Spring Security, your users must use CAS exclusively as an authentication mechanism to your application. In most cases, this is not a problem, as we discussed in the previous section, CAS is designed to proxy authentication requests to one or more authentication stores (similarly to what Spring Security does when delegating to a database or LDAP for authentication). We can see that in this diagram, our application is no longer checking the authentication store to validate users (although a data source is still required to fully populate `UserDetails` of the authenticated user).

When we have completed the basic CAS integration with Spring Security, we could remove the **Log In** link from the home page, and instead enjoy automatic redirection to CAS's login screen when we attempt to access a protected resource. Of course, depending on the application, it can also be beneficial to still allow the user to explicitly log in (so that they can see customized content, and so on).

CAS installation and configuration

CAS has the benefit of having an extremely dedicated team behind it which has done an excellent job of developing both quality software and accurate, straightforward documentation of how to use it. Should you choose to follow along with the examples in this chapter, we'd encourage you to read the appropriate "Getting Started" manual for your CAS platform. We will assume in the examples for this chapter that CAS is deployed at the following location: `http://localhost:8080/cas/`

For this chapter, we'd also suggest running CAS and JBCP Pets in two different instances of Tomcat. This will allow you to keep CAS running while you make code changes to JBCP Pets. We'd recommend configuring the CAS server on port 8080, and the JBCP Pets server on port 8081—our examples in this chapter will reflect this configuration. Additionally, the examples in this chapter were written using the most recent available version of CAS Server, 3.3.5 at the time of this writing.

Be aware that some significant changes to some of the back-end classes were made to CAS in the 3.x timeframe, so if you are on an earlier version of the server, these instructions may be slightly or significantly different for your environment!

Let's go ahead and configure the components required for CAS authentication!

Configuring basic CAS integration

The following diagram illustrates the relationships between the Spring Security components that we'll be configuring to get CAS authentication integrated into our JBCP Pets application. This should help you get your bearings as we run through the bean configuration in the following exercises.
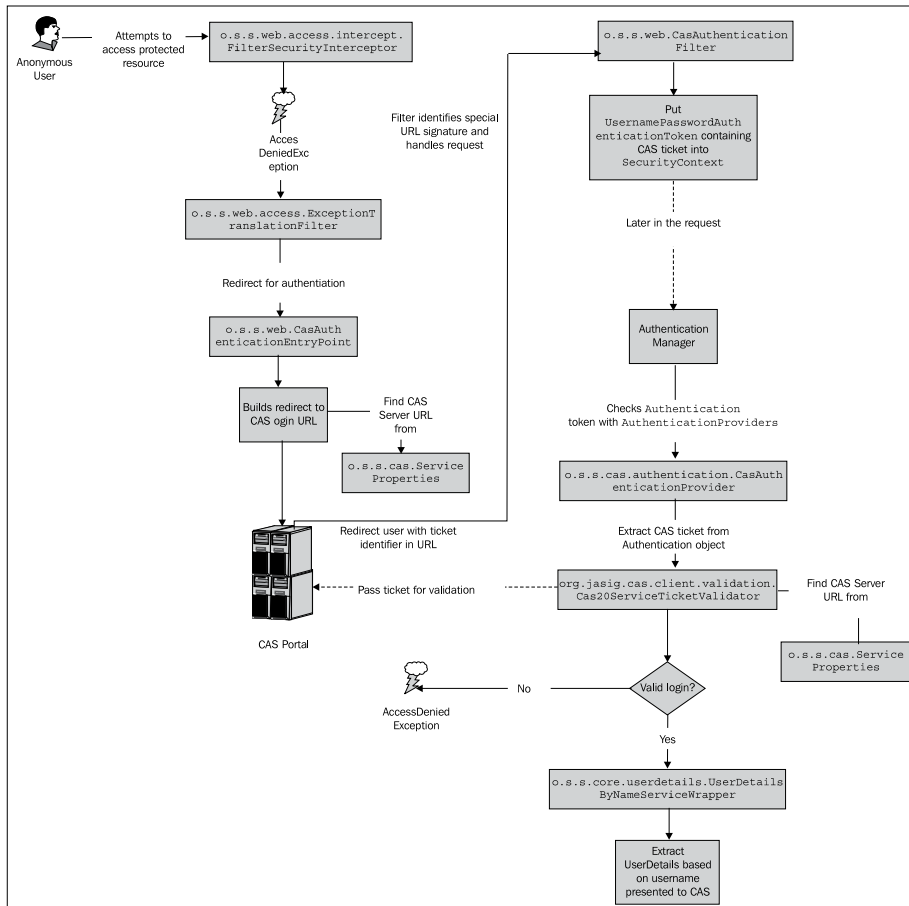
Adding the CasAuthenticationEntryPoint

You may recall from *Chapter 6*, *Advanced Configuration and Extension*, that the implementation of `AuthenticationEntryPoint` is responsible for translating exceptions related to authorization failure into concrete action—redirecting the unauthenticated user to a login page, for example, or presenting the user lacking appropriate authorization with an error page. From the high level architecture diagram, we can see that there is an interception of a user's request to a protected resource which redirects the user to the CAS portal page—this is accomplished through the use of the `o.s.s.cas.web.CasAuthenticationEntryPoint` specifically designed for this purpose.

We'll configure the bean supporting this behavior in `dogstore-base.xml` as follows:

```
<bean id="casAuthEntryPoint" class="org.springframework.security.cas.
web.CasAuthenticationEntryPoint">
  <property name="loginUrl" value="http://localhost:8080/cas/"/>
  <property name="serviceProperties" ref="casService"/>
</bean>
```

Following this, add a reference to the bean in the `security` namespace configuration `dogstore-security.xml` file:

```
<http auto-config="true" …
  entry-point-ref="casAuthEntryPoint">
```

Finally, we will need to declare the `serviceProperties` reference on the `CasAuthenticationEntryPoint`, which refers to an `o.s.s.cas.ServiceProperties` object, declaring the properties of the **service** (application) represented by our application. We'll refer to the URL defined by the CAS authentication filter as follows:

```
<bean id="casService" class="org.springframework.security.cas.
ServiceProperties">
<property name="service" value="http://localhost:8081/JBCPPets/j_
spring_cas_security_check"/>
</bean>
```

We can see that the `ServiceProperties` object plays a role in coordinating data exchange between the various CAS components—it is used as a data object, to store CAS configuration settings that are shared (and are expected to match) by the varying participants in the Spring CAS stack.

> CAS identifies resources needing authentication to access as services. Services are identified by the use of a unique service URL. Within the CAS administrative UI, services can be further defined and configured as desired by CAS administrators.

The `service` property indicates to CAS the service to which the user will be authenticated. Recall that CAS allows selective granting of access per user, per application, based on configuration. We'll examine the particulars of this URL in a moment, when we configure the servlet filter that is expected to process it.

If you start the application at this point and attempt to access a protected resource, you will be immediately redirected to the CAS portal for authentication. An example of a protected page in the application is the **My Account** page—which requires that the user has been authorized with ROLE_USER privilege. The default configuration of CAS allows authentication for any user, where the username is equal to the password—thus, you should be able to log in with a username of `admin` and a password of `admin` (or `guest` / `guest`).

You'll notice, however, that even after login, you will be immediately redirected back to the CAS portal. This is because, although the destination application was able to receive the ticket, it wasn't able to be validated, and as such the `AccessDeniedException` is handled by CAS as a rejection of the ticket.

Enabling CAS ticket verification

Referring to the diagram that we saw earlier in the *Configuring Basic CAS Authentication* section, we can see that Spring Security via the `FilterSecurityInterceptor` is responsible for identifying an unauthenticated request and redirecting the user to CAS. Adding the `CasAuthenticationEntryPoint` has overridden the standard redirect to login page functionality, and provided the expected redirection from the application to the CAS server. Now we need to configure things so that, once authenticated to CAS, the user is properly authenticated to the application.

We remember from *Chapter 8*, *Opening up to OpenID*, that OpenID uses a similar redirection approach, by redirecting unauthenticated users to the OpenID Provider for authentication, and then back to the application with verifiable credentials. CAS differs from OpenID in that, upon the user's return to the application, the application is expected to call back to the CAS server to explicitly validate that the credentials provided are valid and accurate. Contrast this with OpenID that uses the presence of a date-based nonce and preshared key-based signature so that the credentials passed by the OpenID Provider can be independently verified.

The benefit of the CAS approach is that the information passed on from the CAS server to authenticate the user is much simpler—only a single URL parameter is returned to the application by the CAS server. Additionally, the application itself need not track the active or valid tickets, and instead can wholly rely on CAS to verify this information. Much as we saw with OpenID, a servlet filter is responsible for recognizing a redirect from CAS, and processing it as an authentication request. We'll configure the servlet filter first as a Spring Bean in `dogstore-base.xml` as follows:

```
<bean id="casAuthenticationFilter" class="org.springframework.
security.cas.web.CasAuthenticationFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
</bean>
```

We'll then add the custom servlet filter to the `security` namespace configuration in `dogstore-security.xml`:

```
<http auto-config="true" ...>
  ...
  <custom-filter ref="casAuthenticationFilter" position="CAS_
FILTER"/>
</http>
```

Finally, we noted that a reference to the `AuthenticationManager` was required by the `CasAuthenticationFilter`—this is added (if not already present) with the `alias` attribute of the `<authentication-manager>` declaration in `dogstore-security.xml`:

```
<authentication-manager
  alias="authenticationManager">
```

You may have noticed that we referenced the CAS service name as the following URL: `http://localhost:8081/JBCPPets/j_spring_cas_security_check` when we configured the `ServiceProperties` object. As we've seen with other authentication filters, the `/j_spring_cas_security_check` URL indicates a URL signature recognized by the `CasAuthenticationFilter` as a special URL intended to be used only by the redirect response from the CAS server.

Modifying the CAS service URL

You may want to change the CAS Service URL from the default of `/j_spring_cas_security_check`—this can be done by setting the `filterProcessesUrl` property of the CasAuthenticationFilter to any relative URL you want. It can sometimes be beneficial to change the default URL in order to obscure your application's use of Spring Security and/or CAS— while such "security through obscurity" is not foolproof, it will take care of some of the troublesome types of standardized bot attacks that a publically facing site might experience.

The `CasAuthenticationFilter` populates an `Authentication` (a `UsernamePasswordAuthenticationToken`) with special credentials that are recognizable by the next, and final element of a minimal CAS configuration.

Proving authenticity with the CasAuthenticationProvider

If you have been following the logical flow of Spring Security through the rest of this book, hopefully you already know what comes next—the `Authentication` token must be inspected by an appropriate `AuthenticationProvider`. CAS is no different, and as such, the final piece of the puzzle is configuration of an `o.s.s.cas.authentication.CasAuthenticationProvider` within the `AuthenticationManager`.

First, we'll declare the Spring Bean in `dogstore-base.xml` as follows:

```
<bean id="casAuthenticationProvider" class="org.springframework.
security.cas.authentication.CasAuthenticationProvider">
  <property name="ticketValidator" ref="casTicketValidator"/>
  <property name="serviceProperties" ref="casService"/>
  <property name="key" value="jbcp-pets-dogstore-cas"/>
  <property name="authenticationUserDetailsService" ref="authenticatio
nUserDetailsService"/>
</bean>
```

Next, we'll configure a reference to this new `AuthenticationProvider` in `dogstore-security.xml`, where our `<authentication-manager>` declaration resides:

```
<authentication-manager alias="authenticationManager">
   <authentication-provider ref="casAuthenticationProvider"/>
</authentication-manager>
```

If you have any other `AuthenticationProvider` references remaining from prior exercises, please remember to remove them for our work with CAS. Now, we'll need to take care of the other attributes and bean references within the `CasAuthenticationProvider`.

The `ticketValidator` attribute refers to an implementation of the `org.jasig.cas.client.validation.TicketValidator` interface—as we are using CAS 2.0 authentication, we'll declare an `org.jasig.cas.client.validation.Cas20ServiceTicketValidator` instance as follows:

```
<bean id="casTicketValidator" class="org.jasig.cas.client.validation.
Cas20ServiceTicketValidator">
  <constructor-arg value="http://localhost:8080/cas/"/>
</bean>
```

The constructor argument supplied to this class should refer (once again) to the URL used to access the CAS server. You'll note that at this point, we have moved out of the `org.springframework.security` package into `org.jasig`, which is part of the CAS client JAR files. We'll see later in this chapter that the `TicketValidator` interface also has implementations (still within the CAS client JAR files) that support other methods of authentication against CAS such as SAML authentication.

> **Externalize URLs and environment-dependent settings**
>
> If you were configuring a real application with CAS, this is the point where you should be thinking that coding URLs into Spring configuration files is a bad idea. Typically, storage and consistent reference to URLs is pulled out into a separate properties file, with placeholders consistent with the Spring `PropertyPlaceholderConfigurer`. This allows for reconfiguration of environment-specific settings via externalizable properties files without touching the Spring configuration files, and is generally considered good practice.

Next, we see the `key` attribute—this is simply used to validate the integrity of the `UsernamePasswordAuthenticationToken` and can be arbitrarily defined.

Finally, the `authenticationUserDetailsService` attribute refers to an `o.s.s.core.userdetails.AuthenticationUserDetailsService` object that is used to translate the username information from the `Authentication` token to a fully-populated `UserDetails` object, by referring to a `UserDetailsService`. Obviously, this technique would only ever be used when we have confirmed that the integrity of the `Authentication` token has not been compromised. We configure this object with a reference to the `JdbcDaoImpl` implementation of the `UserDetailsService`.

```
<bean id="authenticationUserDetailsService"
 class="org.springframework.security.core.userdetails.
UserDetailsByNameServiceWrapper">
  <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

We may wonder why a `UserDetailsService` isn't directly referenced—it's because there will be additional advanced configuration options later which will allow details from the CAS server to be used to populate the `UserDetails` object.

At this point, we should be able to go start to finish and log in through CAS, and be redirected back to the JBCP Pets application. Excellent job!

Advanced CAS configuration

The CAS authentication framework supports additional configurations and data exchange with the CAS service. We'll explore additional advanced CAS integration capabilities in this section. Where we feel it's important, we'll include relevant CAS configuration instructions, although keep in mind that CAS configuration is complex and well outside the scope of this book!

Retrieval of attributes from CAS assertion

It is possible for CAS to pass information in the response to the ticket validation check with the CAS server, based on the information retrieved by CAS during user authentication. This information is passed in the form of name/value pairs, and can contain any arbitrary data related to the user. We'll demonstrate the capacity to pass arbitrary user attributes, in addition to `GrantedAuthority` information, with the CAS response.

How CAS internal authentication works

Before we jump into CAS configuration, we'll briefly illustrate the standard behavior of CAS authentication processing. The following diagram should help you follow the configuration steps required to allow CAS to talk to our embedded LDAP server:
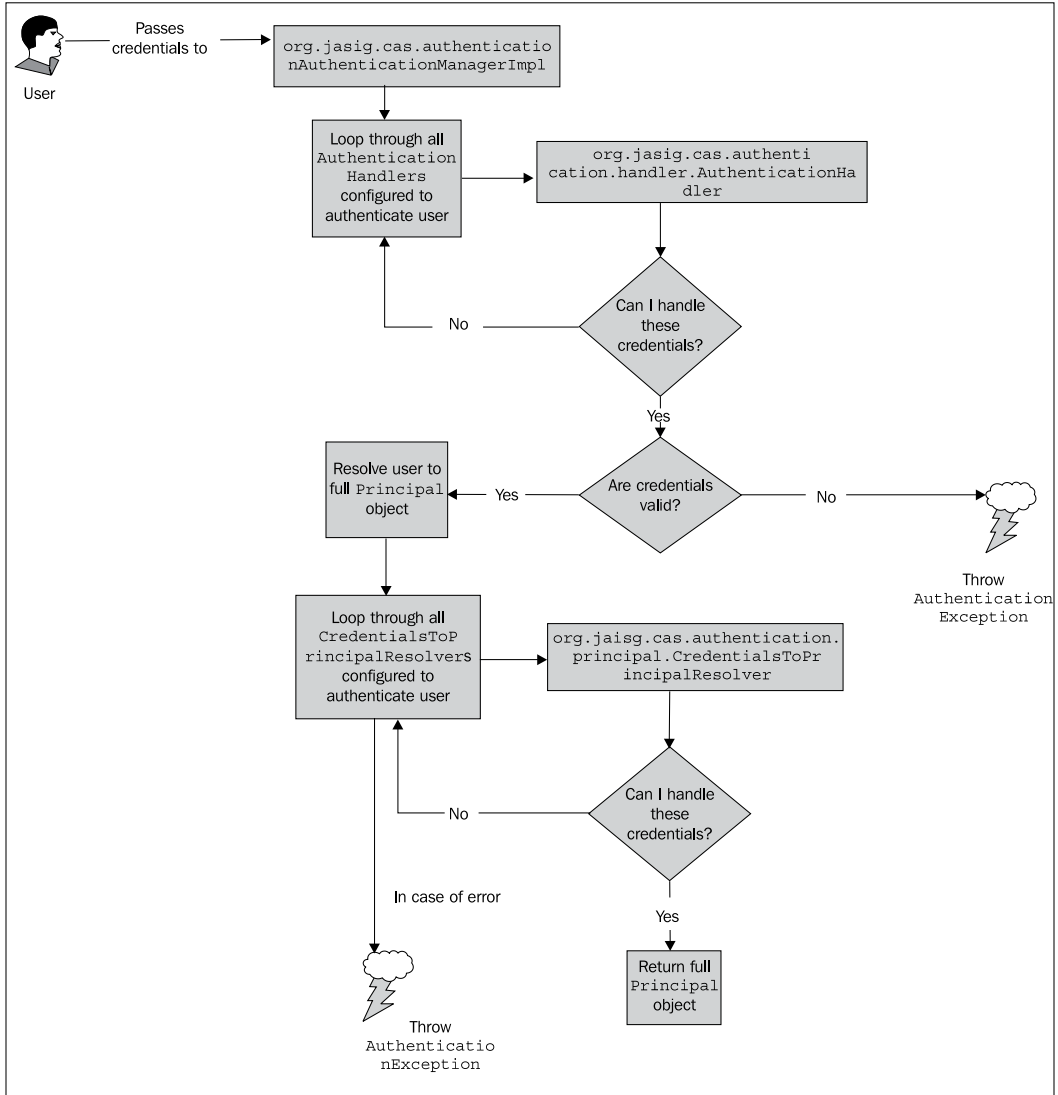
While the previous diagram describes the internal flow of authentication within the CAS server itself, it is likely that if you are implementing integration between Spring Security and CAS, you will also need to adjust the configuration of the CAS server as well. It's important, therefore, that you understand at a high level how CAS authentication works.

The CAS server's `org.jasig.cas.authentication.AuthenticationManager` (not to be confused with the Spring Security class of the same name) is responsible for authenticating the user based on the provided credentials. Much as with Spring Security, the actual processing of the credentials is delegated to one (or more) processing class implementing the `org.jasig.cas.authentication.handler.AuthenticationHandler` interface (we recognize that the analogous interface in Spring Security would be `AuthenticationProvider`).

Finally, a `org.jasig.cas.authentication.principal.CredentialsToPrincipalResolver` is used to translate the credentials passed into a full `org.jasig.cas.authentication.principal.Principal` object (similar behavior in Spring Security occurs in the implementation of `UserDetailsService`).

While not a full review of the behind-the-scenes functionality of the CAS server, this should help you understand the configuration steps in the next several exercises. We encourage you to read the source code for CAS, and consult the web-based documentation available at the JA-SIG CAS wiki: `http://www.ja-sig.org/wiki/display/CAS`.

Configuring CAS to connect to our embedded LDAP server

The `org.jasig.cas.authentication.principal.UsernamePasswordCredentialsToPrincipalResolver` that comes configured by default with CAS doesn't allow us to pass back attribute information and demonstrate this feature of Spring Security CAS integration, so we'd suggest using an implementation which does allow this.

An easy authentication handler to configure and use (especially if you have gone through the prior chapter's LDAP exercises) is an `org.jasig.cas.authentication.handler.AuthenticationHandler`, which communicates with the embedded LDAP server that we used in the previous chapter. We'll lead you through configuration of CAS to return user LDAP attributes in the following guide.

All of the CAS configuration will take place in the `WEB-INF/deployerConfigContext.xml` file of the CAS installation, and will typically involve inserting class declarations into configuration file segments that already exist.

> If the contents of this file look familiar to you, it's because CAS uses the Spring Framework for its configuration, just like JBCP Pets! We'd recommend using a good IDE with handy reference to the CAS source code if you want to dig into what these configuration settings do. Remember that in this section, and all sections where we refer to `WEB-INF/deployerConfigContext.xml`, we are referring to the CAS installation and not JBCP Pets.

First, we'll add a new `AuthenticationHandler` in place of the `SimpleTestUsernamePasswordAuthenticationHandler`, which will attempt to bind the user to LDAP (just as we did in *Chapter 9, LDAP Directory Services*). The `AuthenticationHandler` will be placed in the `authenticationHandlers` property of the `authenticationManager` bean:

```
<property name="authenticationHandlers">
  <list>
<!-- ... -->
  <bean class="org.jasig.cas.adaptors.ldap.
BindLdapAuthenticationHandler">
    <property name="filter" value="uid=%u" />
    <property name="searchBase" value="ou=Users,dc=jbcppets,dc=com" />
    <property name="contextSource" ref="contextSource" />
  </bean>
```

Don't forget to remove the reference to the `SimpleTestUsernamePasswordAuthenticationHandler`, or at least move its definition to after that of the `BindLdapAuthenticationHandler`, otherwise your CAS authentication will not use LDAP and instead use the stub handler!

You'll note the bean reference to a `contextSource` bean—this defines the `org.springframework.ldap.core.ContextSource` implementation, which CAS will use to interact with LDAP (yes, CAS uses Spring LDAP as well). We'll define this at the end of the file, as follows:

```
<bean id="contextSource" class="org.springframework.ldap.core.support.
LdapContextSource">
 <property name="urls">
  <list>
   <value>ldap://127.0.0.1:33389</value>
  </list>
 </property>
 <property name="userDn" value="uid=ldapadmin,ou=Administrators,ou=Use
rs,dc=jbcppets,dc=com"/>
 <property name="password" value="password"/>
 <property name="baseEnvironmentProperties">
```

```
    <map>
     <entry>
       <key>
        <value>java.naming.security.authentication</value>
       </key>
       <value>simple</value>
     </entry>
    </map>
   </property>
  </bean>
```

Much as we had discussed when configuring Spring Security to attach to an external (non-embedded) LDAP server, CAS requires a user DN for an administrative user in order to bind to the LDAP directory in the first place. In this case, we use the administrative DN that we defined in the `JBCPPets.ldif` bootstrap exercise in Chapter 9. The URL of `ldap://127.0.0.1:33389` contains the port 33389, which is used by the Spring Security embedded LDAP server by default. As we discussed in Chapter 9, in a production configuration, you would most likely use LDAPS to ensure network security of CAS's LDAP requests.

Finally, we'll need to configure a new `org.jasig.cas.authentication. principal.CredentialsToPrincipalResolver`, which is responsible for translating the credentials that the user has provided (that CAS has already authenticated using the `BindLdapAuthenticationHandler`) to a full `org.jasig.cas.authentication. principal.Principal` authenticated principal. You'll notice many configuration options in this class, which we'll skim over, but which you are welcome to dive into as you explore CAS further.

Add the following bean definition inline to the `credentialsToPrincipalResolvers` property of the CAS `authenticationManager` bean:

```
<property name="credentialsToPrincipalResolvers">
  <list>
    <bean class="org.jasig.cas.authentication.principal.
CredentialsToLDAPAttributePrincipalResolver">
      <property name="credentialsToPrincipalResolver">
        <bean class="org.jasig.cas.authentication.principal.
UsernamePasswordCredentialsToPrincipalResolver" />
      </property>
      <property name="filter" value="(uid=%u)" />
      <property name="principalAttributeName" value="uid" />
      <property name="searchBase" value="ou=Users,dc=jbcppets,dc=com"
/>
      <property name="contextSource" ref="contextSource" />
      <property name="attributeRepository" ref bean="attributeReposit
ory" />
    </bean>
```

You'll notice that, as with Spring Security LDAP configuration, much of the same behavior exists in CAS with principals being searched on property matches below a subtree of the directory, based on a DN.

Note that we haven't yet configured the `attributeRepository` ourselves, which should refer to an implementation of `org.jasig.services.persondir.IPersonAttributeDao`. CAS ships with a default configuration that includes a simple implementation of this interface, `org.jasig.services.persondir.support.StubPersonAttributeDao`, which will be sufficient until we configure LDAP-based attributes in a later exercise.

So, now we've configured basic LDAP authentication in CAS. At this point, you should be able to restart CAS, start JBCP Pets (if it's not already running), and authenticate using any of the LDAP users we created in Chapter 9 (username `ldapguest` with password `password` is a good candidate). You'll get an ugly 403 Access Denied page when returning to the application, however, due to the fact that the users present in LDAP aren't found in the database, where our `AuthenticationUserDetailsService` is pointing. Let's remedy this situation now.

Getting UserDetails from a CAS assertion

When we first set up CAS integration with Spring Security, we configured a `UserDetailsByNameServiceWrapper`, which simply translated the username presented to CAS into a `UserDetails` object from the `UserDetailsService` that we had referenced (in our case, it was the `JdbcDaoImpl`). Now that CAS is referencing the LDAP server, we could set up an `LdapUserDetailsService` as we discussed at the tail end of Chapter 9, and things would work just fine.

Instead, we'll experiment with another capability of the Spring Security CAS integration, the ability to populate a `UserDetails` from the CAS assertion itself! This is actually as simple as switching the `AuthenticationUserDetailsService` implementation to the `o.s.s.cas.userdetails.GrantedAuthorityFromAssertionAttributesUserDetailsService`, whose job it is to read the CAS assertion, look for a certain attribute, and map the value of that attribute directly to `GrantedAuthority` for the user. Let's assume that there is an attribute entitled `role` which will be returned with the assertion. We'll simply configure a new `authenticationUserDetailsService` bean in `dogstore-base.xml`:

```
<bean id="authenticationUserDetailsService"
  class="org.springframework.security.cas.userdetails.
GrantedAuthorityFromAssertionAttributesUserDetailsService">
  <constructor-arg>
    <array>
      <value>role</value>
```

```
      </array>
    </constructor-arg>
  </bean>
```

Next, we'll add some minor debugging functionality to allow us to review the contents of the assertion.

Examining the CAS assertion

To assist in our review of the information that CAS provides back to the JBCP Pets application, we'll amend the `AccountController` to show some information about the logged-in CAS user and the information that CAS provided us about them. First, we'll add a simple URL handler method to `AccountController`:

```
@RequestMapping(value="/account/viewCasUserProfile.
do",method=RequestMethod.GET)
public void showViewCasUserProfilePage(ModelMap model) {
  final Authentication auth = SecurityContextHolder.getContext().
getAuthentication();
  model.addAttribute("auth", auth);
  if(auth instanceof CasAuthenticationToken) {
    model.addAttribute("isCasAuthentication", Boolean.TRUE);
  }
}
```

Next, we'll add a JSP which will display some interesting information about the `CasAuthenticationToken` representing a user's authenticated CAS login. This will be placed in `WEB-INF/views/account/viewCasUserProfile.jsp`.

```
<!-- Common Header and Footer Omitted -->
<h1>View Profile</h1>
<p>
  Some information about you, from CAS:
</p>
<ul>
  <li><strong>Auth:</strong> ${auth}</li>
  <li><strong>Username:</strong> ${auth.principal}</li>
  <li><strong>Credentials:</strong> ${auth.credentials}</li>
  <c:if test="${isCasAuthentication}">
    <li><strong>Assertion:</strong> ${auth.assertion}</li>
    <li><strong>Assertion Attributes:</strong>
    <c:forEach items="${auth.assertion.attributes}" var="attr">
      ${attr.key}:${attr.value}<br />
    </c:forEach>
    </li>
```

```
    <li><strong>Assertion Attribute Principal:</strong> ${auth.
  assertion.principal}</li>
    <li><strong>Assertion Principal Attributes:</strong>
    <c:forEach items="${auth.assertion.principal.attributes}"
  var="attr">
      ${attr.key}:${attr.value}<br />
    </c:forEach>
    </li>
  </c:if>
</ul>
```

Add a simple link to the account home page, in `WEB-INF/views/account/home.jsp`:

```
<h1>Welcome to Your Account</h1>
<!-- omitted -->
<ul>
  <li><a href="viewCasUserProfile.do">View CAS User Profile</a></li>
```

Finally, we'll have to (temporarily) disable authorization checks for this page, until we get assertion attribute-based authorization working. Do this with a simple adjustment in `dogstore-security.xml`, so that logged in users of any `GrantedAuthority` can access this page, and the **My Account** page:

```
<intercept-url pattern="/home.do" access="permitAll"/>
<intercept-url pattern="/account/home.do" access="!anonymous"/>
<intercept-url pattern="/account/view*Profile.do"
access="!anonymous"/>
<intercept-url pattern="/account/*.do" access="hasRole('ROLE_USER')"/>
```

Once you've completed these minor UI updates, restart the JBCP Pets application, and try accessing this page. You'll see that it provides a lot of information about the information contained within the assertion.

Mapping LDAP attributes to CAS attributes

The final piece of the puzzle requires us to map LDAP attributes to attributes in the CAS assertion (including the `role` attribute that we're expecting to contain the user's `GrantedAuthority`).

We'll add another bit of configuration to the CAS `deployerConfigContext.xml`. This new bit of configuration is required to instruct CAS how to map attributes from the CAS `Principal` object to the CAS `IPersonAttributes` object, which will ultimately be serialized as a part of the ticket validation. This bean configuration should replace the bean of the same name, `attributeRepository`.

```
<bean id="attributeRepository" class="org.jasig.services.persondir.
support.ldap.LdapPersonAttributeDao">
  <property name="contextSource" ref="contextSource" />
  <property name="requireAllQueryAttributes" value="true" />
  <property name="baseDN" value="ou=Users,dc=jbcppets,dc=com" />
  <property name="queryAttributeMapping">
    <map>
      <entry key="username" value="uid" />
    </map>
  </property>
  <property name="resultAttributeMapping">
    <map>
      <entry key="cn" value="FullName" />
      <entry key="sn" value="LastName" />
      <entry key="description" value="role" />
    </map>
  </property>
</bean>
```

The functionality behind the scenes here is definitely confusing—essentially, the purpose of this class is to map the `Principal` back to the LDAP directory (this is the `queryAttributeMapping` property, mapping the `username` field of the `Principal` to the `uid` attribute in the LDAP query). The `baseDN` provided is searched using the LDAP query `(uid=ldapguest)`, and attributes are read from the matching entry. The attributes are mapped back to the `Principal` using the key/value pairs in the `resultAttributeMapping` property—we recognize the LDAP `cn` and `sn` attributes being mapped to meaningful names, and the `description` attribute being mapped to the `role` that our `GrantedAuthorityFromAssertionAttributesUserDetailsService` will be looking for.

Part of the complexity here comes from the fact that a portion of this functionality is wrapped up in a separate project, Person Directory (`http://www.ja-sig.org/wiki/display/PD/Home`), which is intended to aggregate multiple sources of information about a person into a single view. The design of Person Directory is such that it is not directly tied to the CAS server, and could be reused as part of other applications. The downside of this design choice is that it makes some aspects of CAS configuration more complex than it initially seems should be required.

> **Some Gotchas on LDAP attribute mapping in CAS**
>
> We would love to set up the same type of query in LDAP as we used
> with Spring Security LDAP in Chapter 9, to be able to map a `Principal`
> to a full LDAP distinguished name, and then to use that DN to look up
> group membership by matching on the basis of `uniqueMember` attribute
> of a `groupOfUniqueNames` entry. Unfortunately, the CAS LDAP code
> doesn't yet have this flexibility, leading to the conclusion that more
> advanced LDAP mapping will require extensions to base classes in CAS.

Whoa there! Confused yet? We admit this is pretty confusing to us too, as the various
bits of CAS convert back and forth in several different ways between LDAP data
and CAS data. The CAS mailing lists and community wiki (`http://www.ja-sig.`
`org/wiki/display/CASUM/Home`) are great places to read about other peoples'
experiences with the product and ask questions to a knowledgeable audience.

Finally, returning the attributes in the CAS assertion

Unfortunately, the CAS 2.0 protocol does not specify a standard format for return
of attribute data. There have been several attempts, as documented in the CAS JIRA
bug tracking system, to implement this functionality, but these have been rejected, as
the CAS 2.0 protocol is stable and is probably not going to change in the near future.

That said, many users do extend the CAS response to include attribute data anyway.
Ultimately, CAS's response to the client application's ticket validation request is
rendered by a JSP, and as such, this is fairly easy to modify in your CAS installation
to include a response, which conforms to the attribute parsing expected by the
`Cas20ServiceTicketValidator`. In your CAS deployment, edit `WEB-INF/view/`
`jsp/protocol/2.0/casServiceValidationSuccess.jsp` and add the following:

```
<cas:authenticationSuccess>
  <cas:user>${fn:escapeXml(assertion.chainedAuthentications[fn:
length(assertion.chainedAuthentications)-1].principal.id)}</cas:user>
  <cas:attributes>
  <c:forEach var="attr"
  items="${assertion.chainedAuthentications[fn:length(assertion.
chainedAuthentications)-1].principal.attributes}"
  varStatus="loopStatus" begin="0"
  end="${fn:length(assertion.chainedAuthentications[fn:
length(assertion.chainedAuthentications)-1].principal.attributes)-1}"
  step="1">
  <cas:${fn:escapeXml(attr.key)}>${fn:escapeXml(attr.value)}</
cas:${fn:escapeXml(attr.key)}>
  </c:forEach>
  </cas:attributes>
```

This will result in the following CAS response format:

```
<cas:serviceResponse xmlns:cas="http://www.yale.edu/tp/cas">
  <cas:authenticationSuccess>
    <cas:user>ldapguest</cas:user>
    <cas:attributes>
      <cas:FullName>LDAP Guest</cas:FullName>
      <cas:role>ROLE_USER</cas:role>
      <cas:LastName>Guest</cas:LastName>
    </cas:attributes>
  </cas:authenticationSuccess>
</cas:serviceResponse>
```

With these changes in place and both CAS and JBCP Pets restarted, you should be able to log in as `ldapguest` and access areas of the application, which are protected by `ROLE_USER` security. Additionally, you should see the **View CAS Profile** page displaying the attributes returned via the CAS assertion.

> Take note—we can see that the attribute names provided in the `LdapPersonAttributeDao.resultAttributeMapping` are dropped directly in the XML of the CAS response. This means that they can't be entirely arbitrary, and must conform to valid XML element naming conventions (for example, they cannot contain spaces). If you want to change this behavior, you may require more complicated JSP code than that which is demonstrated here.

Great job—this is a lot of configuration work across two complex products. Take a break and have a cup of coffee!

Note that an alternative approach that some CAS developers choose to take is to extend `Cas20ServiceTicketValidator` (on the Spring Security side of the conversation) to perform custom processing of the ticket response returned by CAS.

Alternative Ticket authentication using SAML 1.1

SAML is a standard, cross-platform protocol for identity verification through structured XML assertions. SAML is supported by a wide variety of products, including CAS (in fact, we will look at support for SAML within Spring Security itself in a later chapter).

The SAML security assertion XML dialect solves some of the issues with attribute passing using the CAS response protocol we previously described. Happily, switching between CAS ticket validation and SAML ticket validation is as simple as changing the `TicketValidator` implementation configured in `dogstore-base.xml`. Add a bean as follows:

```
<bean id="samlTicketValidator" class="org.jasig.cas.client.validation.
Saml11TicketValidator">
  <constructor-arg value="http://localhost:8080/cas/"/>
</bean>
```

Then update the `CasAuthenticationProvider` to use this SAML `TicketValidator` instead:

```
<bean id="casAuthenticationProvider" class="org.springframework.
security.cas.authentication.CasAuthenticationProvider">
  <property name="ticketValidator" ref="samlTicketValidator"/>
  <property name="serviceProperties" ref="casService"/>
```

Restart the JBCP Pets application, and SAML responses will now be in use for ticket validation. Note that CAS ticket responses are not logged by Spring Security CAS classes, so you will need to either enable logging for the JA-SIG CAS client classes (enable logging for `org.jasig` in log4j) or examine the differences in the responses using a debugger.

In general, it's recommended that SAML ticket validation be used over CAS 2.0 ticket validation, as it adds more non-repudiation features, including timestamp validation, and solves the attribute problem in a standard way.

How is Attribute Retrieval useful?

Remember that CAS provides a layer of abstraction for our application, removing the ability for our application to directly access the user repository and instead, forcing all such access to be performed through CAS as a proxy.

This is extremely powerful! It means that our application no longer cares what kind of repository users are stored in, nor does it have to worry about the details of how to access them—simply confirming authentication with CAS is sufficient to prove that a user should be able to access our application. For system administrators, this means that, should an LDAP server be renamed, moved, or otherwise adjusted, they only need to reconfigure it in a single location – CAS. Centralizing access through CAS allows for a high level of flexibility and adaptability in the overall security architecture of the organization.

Extend this story to the usefulness of attribute retrieval from CAS—now, all applications authenticated through CAS have the same view of a user, and can consistently display information across any CAS-enabled environment.

Be aware that, once authenticated, Spring Security CAS does not re-query the CAS server unless the user is required to re-authenticate. This means that attributes and other user information stored locally in the application in the user's `Authentication` object may become stale over time and possibly out of sync with the source CAS server. Take care to set session timeouts appropriately to avoid this potential issue!

Additional CAS capabilities

CAS offers additional advanced configuration capabilities outside of those which are exposed through the Spring Security CAS wrappers. Some of these include the following:

- Providing transparent single sign-on for users who are accessing multiple CAS-secured applications within a configurable (on the CAS server) time window. Applications can force users to authenticate to CAS by setting the `renew` property to true on the `TicketValidator`—you may want to conditionally set this property in custom code, in the event where the user is attempting to access a highly secured area of the application.

- Providing ticket proxy capabilities to secondary applications which may not be able to directly access CAS. When requesting a ticket from CAS, the web application may request to grant tickets by proxy to a secondary application. You may read more about this capability at the CAS website (`http://www.jasig.org/cas/proxy-authentication`).

- Coordinated single sign-out among participating applications where the user has an active CAS session (note this is not directly supported by Spring Security, but is supported by the CAS client through a combination of `HttpSessionListener` and servlet filter).

We'd encourage you to explore the full capabilities of the CAS client and server, as well as ask questions to the helpful folks in the JA-SIG community forums!

Summary

In this chapter, we learned about the Central Authentication System (CAS) Single Sign-On portal, how it can be integrated with Spring Security and we also covered the following:

- The CAS architecture and communication paths between actors in a CAS enabled environment
- The benefits of CAS enabled applications for application developers and system administrators
- Configuring JBCP Pets to interact with a basic CAS installation
- Updating CAS to interact with LDAP and sharing LDAP data with our CAS enabled application
- Implementing attribute exchange with CAS using both a modified CAS 2.0 protocol, as well as the industry standard SAML protocol

We hope this chapter was an interesting introduction to the world of single sign-on. There are many other single sign-on systems in the marketplace, mostly commercial, but CAS is definitely one of the leaders of the open source SSO world, and an excellent platform to build out SSO capability in any organization.

In the next chapter, we'll move back to standard authentication, but this time we will remove usernames and passwords altogether and instead rely on a completely new authentication mechanism. Excited? So are we!

11
Client Certificate Authentication

Although username and password authentication is extremely common, as we discussed in *Chapter 1, Anatomy of an Unsafe Application* and *Chapter 2, Getting Started with Spring Security*; forms of authentication exist that allow users to present different types of credentials. Spring Security caters to these requirements as well, and in this chapter we'll move beyond form-based authentication to explore authentication using trusted client-side certificates.

During the course of this chapter we'll:

- Learn how client certificate authentication is negotiated between the user's browser and a compliant server
- Configure Spring Security to authenticate users with client certificates
- Understand the architecture of client certificate authentication in Spring Security
- Explore advanced configuration options related to client certificate authentication
- Review pros, cons, and common troubleshooting steps when dealing with client certificate authentication

How Client Certificate authentication works

Client certificate authentication requires a request for information from the server, and a response from the browser, to negotiate a trusted authentication relationship between the user (and their browser) and the server application. This trusted relationship is built through the use of the exchange of trusted and verifiable credentials, known as **certificates**.

Unlike much of what we have seen to this point, with client certificate authentication, the servlet container or application server itself is typically responsible for negotiating the trust relationship between the browser and the server, by requesting a certificate, evaluating it, and accepting it as valid.

Client certificate authentication is also known as **mutual authentication**, and is part of the **Secure Sockets Layer** (**SSL**) protocol, and its successor, **Transport Layer Security** (**TLS**). As mutual authentication is part of the SSL and TLS protocols, it follows that an HTTPS connection (secured with SSL or TLS) is required in order to make use of client certificate authentication. For more details on SSL/TLS support in Spring Security, please refer to our discussion and implementation of SSL/TLS in *Chapter 4, Securing Credential Storage*. Setting up SSL/TLS in Tomcat (or the application server you have been using to follow along with the examples) is required to implement client certificate authentication. As in Chapter 4, we will refer to SSL/TLS as SSL for the remainder of this chapter.

The following sequence diagram illustrates the interaction between the client browser and the web server, when negotiating an SSL connection and validating the trust of a client certificate used for mutual authentication.
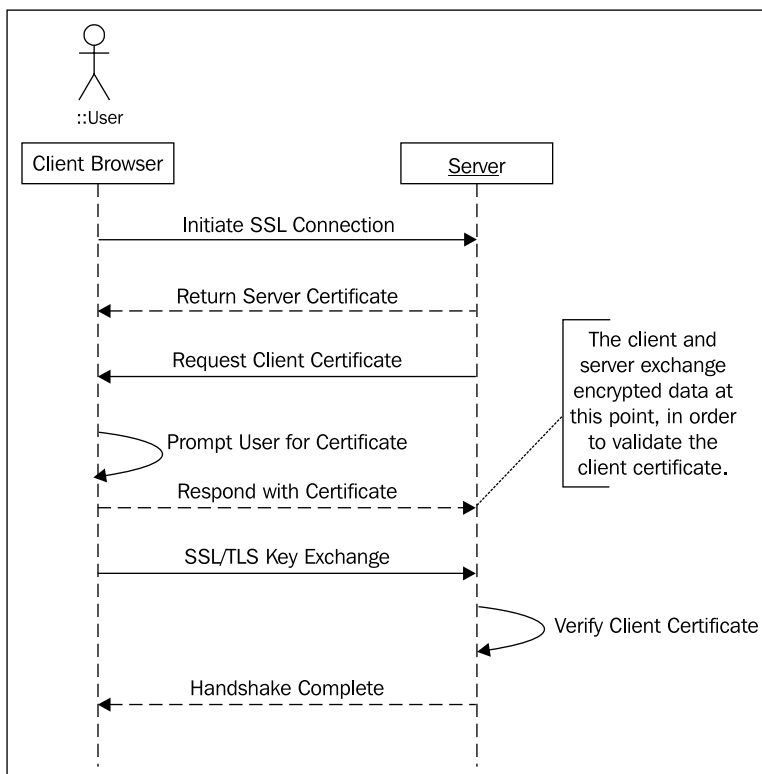
We can see that the exchange of two certificates, the server certificate and the client certificate, provides authentication that both parties are known and can be trusted to continue their conversation securely. In the interest of clarity, we omit some details of the SSL handshake and trust checking of the certificates themselves; however, you are encouraged to do further reading in the area of the SSL and TLS protocols, and certificates in general, as many good reference guides on these subjects exist. RFC 5246, *The Transport Layer Security (TLS) Protocol V1.2* (`http://tools.ietf.org/ html/rfc5246`), is a good place to begin reading about client certificate presentation, and if you'd like to get into more detail, *SSL and TLS: Designing and Building Secure Systems*, by Eric Rescorla, is an incredibly detailed review of the protocol and its implementation.

An alternative name for client certificate-based authentication is **X.509** authentication. The term X.509 is derived from the X.509 standard, originally published by the ITU-T organization, for use in directories based on the X.500 standard (the origins of LDAP, as you may recall from *Chapter 9, LDAP Directory Services*). Later, this standard was adapted for use in securing internet communications.

We mention this here because many of the classes in Spring Security related to this subject refer to X.509. Remember that X.509 doesn't define the mutual authentication protocol itself, but instead defines the format and structure of certificates and the encompassing trusted **certificate authorities**.

Setting up a Client Certificate authentication infrastructure

Unfortunately for you as an individual developer, being able to experiment with client certificate authentication requires some non-trivial configuration and setup prior to the relatively easy integration with Spring Security. As these setup steps tend to cause a lot of problems for first-time developers, we felt it was important to walk you through them.

We assume that you are using a local, self-signed server certificate, and self-signed client certificates, as well as Apache Tomcat. This is typical of most development environments; however, it's possible that you may have access to a valid server certificate, a certificate authority (CA), or another application server. If this is the case, you may use these setup instructions as guidelines, and configure your environment in an analogous manner. Please refer to the SSL setup instructions in Chapter 4 for assistance in configuring Tomcat and Spring Security to work with SSL in a standalone environment.

Understanding the purpose of a public key infrastructure

While this chapter focuses on setting up a self-contained development environment for the purposes of learning and education, in most cases where you are integrating Spring Security into an existing client certificate-secured environment, there will be a significant amount of infrastructure (usually a combination of hardware and software) in place to provide functionality such as certificate granting and management, user self-service, and revocation. Environments of this type define a public key infrastructure—a combination of hardware, software, and security policies that result in a highly secure, authentication-driven network ecosystem.

In addition to being used for web application authentication, certificates or hardware devices in these environments can be used for secure, non-repudiated email (using S/MIME), network authentication, and even physical building access (using PKCS 11-based hardware devices).

While the management overhead of such an environment can be high (and requires both IT and process excellence to implement well), it is arguably one of the most secure possible operating environments for technology professionals.

Creating a client certificate key pair

The self-signed client certificate is created in the same way as the self-signed server certificate is created, by generating a key pair using the `keytool` command. A client certificate key pair differs in that it requires the key store to be available to the web browser, and requires the client's public key to be loaded into the server's **trust store** (we'll explain what this is in a moment)

Create the client key pair as follows:

```
keytool -genkeypair -alias jbcpclient -keyalg RSA -validity 365 -
keystore jbcppets_clientauth.p12 -storetype PKCS12
```

When prompted to set up the first and last name (the **common name**, or CN, portion of the owner's DN) for the client certificate, ensure that the answer to the first prompt matches a user that we have set up in our Spring Security JDBC store, for example `admin`:

```
What is your first and last name?
  [Unknown]:  admin
... etc
Is CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US
correct?
  [no]:  yes
```

We'll see why this is important when we configure Spring Security to access the information from the certificate-authenticated user. We have one final step before we can set up certificate authentication within Tomcat.

Configuring the Tomcat trust store

Recall that the definition of a key pair includes both a private and public key. Much as with SSL certificates verifying and securing server communication, the validity of the client certificate needs to be verified against the certifying authority which created it.

As we have created our own self-signed client certificate using the `keytool` command, the Java VM will not implicitly trust it as having been assigned by a trusted certificate authority.

As such, we will need to force Tomcat to recognize the certificate as a trusted certificate. We do this by exporting the public key from the key pair and adding it to the Tomcat trust store.

First, we'll export the public key to a standard certificate file, named `jbcppets_clientauth.cer` as follows:

```
keytool -exportcert -alias jbcpclient -keystore jbcppets_clientauth.
p12 -storetype PKCS12 -storepass password -file
jbcppets_clientauth.cer
```

Next, we'll import the certificate into the trust store (this will create the trust store, but in a typical deployment scenario, you'd probably already have some other certificates in the trust store):

```
keytool -importcert -alias jbcpclient -keystore tomcat.truststore -
file jbcppets_clientauth.cer
```

This will create the trust store called `tomcat.truststore` and prompt you for a password. You'll also see some information about the certificate and, finally, be asked to confirm that you do trust the certificate:

```
Owner: CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US
Issuer: CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US
Serial number: 4b3fb3d9
Valid from: Sat Jan 02 16:00:09 EST 2010 until: Sun Jan 02 16:00:09
EST 2011
Certificate fingerprints:
        MD5:  02:69:16:3B:D7:C2:74:9E:F7:FD:18:C9:C5:E4:C8:94
        SHA1: 65:57:94:6D:D2:83:7E:51:19:CF:58:94:ED:43:11:F6:AC:D0:
FB:EC
        Signature algorithm name: SHA1withRSA
        Version: 3
Trust this certificate? [no]:  yes
```

Copy the new `tomcat.truststore` file to the `conf` directory of the Tomcat server you are using. One final configuration step before we're ready to go!

What's the difference between a Key Store and a Trust Store?

The **Java Secure Socket Extension** (**JSSE**) documentation defines a key store as a storage mechanism for private keys and their corresponding public keys. The key store (containing key pairs) is used to encrypt or decrypt secure messages, and so on. The trust store is intended to store only public keys for trusted communication partners, when verifying identity (such as how we see the trust store used in certificate authentication). In many common administration scenarios, however, the key store and trust store are combined into a single file (in Tomcat, this would be done through the use of the `keystoreFile` and `truststoreFile` attributes of the `Connector`). The format of the files themselves can be exactly the same (really, each file can be any JSSE supported keystore format, including Java Key Store / JKS, PKCS 12, and so on).

Finally, we'll need to point Tomcat at the trust store, and enable client certificate authentication. This is done by adding three additional attributes to the SSL `Connector` in the Tomcat `server.xml` file:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
  maxThreads="150" scheme="https" secure="true"
  sslProtocol="TLS"
  keystoreFile="conf/tomcat.keystore"
  keystorePass="password"
  truststoreFile="conf/tomcat.truststore"
  truststorePass="password"
  clientAuth="true"
  />
```

This should be the remaining configuration required to trigger Tomcat to request a client certificate when any SSL connection is made. At this point, we should be able to restart Tomcat.

There's also a way to configure Tomcat to optionally use client certificate authentication—we'll enable this later in the chapter. For now, we require the use of client certificates to even connect to the Tomcat server in the first place. This makes it easier to diagnose whether or not you have set this up correctly!

The final step is to import the certificate into the client browser.

Importing the certificate key pair into a browser

Depending on what browser you are using, the process of importing a certificate may differ. We will provide instructions for Firefox and Internet Explorer here, but if you are using another browser, please consult its help or your favorite search engine for assistance.

Using Firefox

Follow these steps to import the key store containing the client certificate key pair.

1. Open the Tools menu
2. Click on the **Options...** menu item.
3. Click on the **Advanced** button/icon.
4. Click on the **Encryption** tab.
5. Click on the **View Certificates** button. The **Certificate Manager** should open up.
6. Click on the **Your Certificates** tab.
7. Click on the **Import...** button.
8. Browse to the location where you saved the `jbcppets_clientauth.p12` file and select it .
9. You will need to enter the password that you used when you created the file.

The client certificate should be imported, and you should see it in the list.

Using Internet Explorer

As Internet Explorer is tightly integrated into the Windows OS, it's a bit easier to import the key store.

1. Double-click the `jbcppets_clientauth.p12` file in Windows Explorer. The **Certificate Import Wizard** should start.
2. Click on **Next** and accept defaults until you are prompted for the certificate password.
3. Enter the certificate password and click on **Next**.
4. Accept the default **Automatically select the certificate store option** and click on **Next**.
5. Click on **Finish**.

To verify that the certificate was installed correctly, you will need to follow another series of steps.

1. Open the **Tools** menu in Internet Explorer.
2. Click on the **Internet Options** menu item.
3. Click on the **Content** tab.
4. Click on the **Certificates** button.
5. Click on the **Personal** tab, if it is not already selected. You should see the certificate listed here.

Wrapping up testing

You should now be able to connect to the JBCP Pets site using the client certificate. If all is set up correctly, you should be prompted for a certificate when you attempt to access the site—in Firefox, the certificate is displayed as follows:
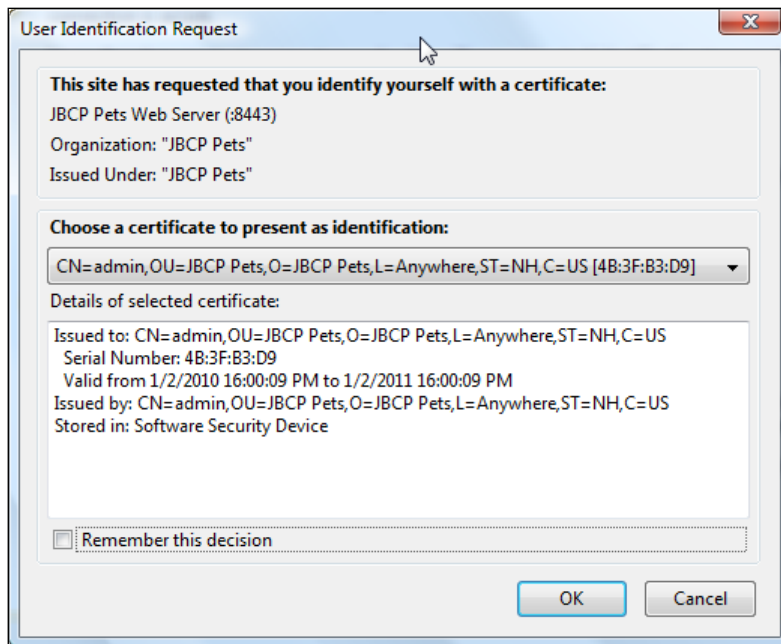
You'll however notice that if you attempt to access a protected section of the site, such as the **My Account** section, you'll be redirected to the login page. This is because we haven't yet configured Spring Security to recognize the information in the certificate—at this point, all the negotiation between client and server has stopped at the Tomcat server itself.

Troubleshooting Client Certificate authentication

Unfortunately, if we said that getting client certificate authentication configured correctly for the first time you tried without anything going wrong was easy, then we'd be lying to you. The fact is, although this is a great and very powerful security apparatus, it is poorly documented by both the browser and web server manufacturers, and the error messages, when present, can be confusing at best, and misleading at worst.

Remember that at this point we have not involved Spring Security in the equation at all, so a debugger will most probably not help you (unless you have the Tomcat source code handy). Some common errors and things to check are as follows:

- You aren't prompted for a certificate when you access the site. There are many possible causes for this, and this can be the most puzzling problem to try to solve. Here are some things to check:
  - ° Ensure the certificate has been installed in the browser client you are using. Sometimes you need to restart the whole browser (close all windows), if you attempted to access the site previously and were rejected.
  - ° Ensure you are accessing the SSL port for the server (typically 8443 in a development setup), and have selected the `https` protocol in your URL. Client certificates are not presented for insecure browser connections. Make sure the browser also trusts the server SSL certificate, even if you have to force it to trust a self-signed certificate.
  - ° Ensure you have added the `clientAuth` directive to your Tomcat configuration (or equivalent for whatever application server you are using).
  - ° If all else fails, use a network analyzer or packet sniffer, such as Wireshark (`http://www.wireshark.org/`) or Fiddler2 (`http://www.fiddler2.com/`) to review the traffic and SSL key exchange over the wire (check with your IT department first—many companies do not allow tools of this kind on their networks).

- If you are using a self-signed client certificate, make sure the public key has been imported into the server's trust store. If you are using a CA-assigned certificate, make sure the CA is trusted by the JVM, or the CA certificate is imported into the server's trust store.

- Internet Explorer in particular does not report details of client certificate failures at all (it simply reports a generic **Page Cannot be Displayed** error). Use Firefox for diagnosing whether an issue you are seeing is related to client certificates.

Configuring Client Certificate authentication in Spring Security

Unlike authentication mechanisms that we have utilized thus far, the use of client certificate authentication results in the user's request having been **pre-authenticated** by the server. As the server (Tomcat) has already established that the user has provided a valid and trustworthy certificate, Spring Security can simply trust this assertion of validity.

An important component of the secure login process is still missing, that is, authorization of the authenticated user. This is where our configuration of Spring Security comes in—we must add a component to Spring Security that will recognize the certificate authentication information from the user's HTTP session (populated by Tomcat), and then validate the presented credentials against the Spring Security `UserDetailsService`. As with all `UserDetailsService` invocations, this will result in the determination of whether or not the user declared in the certificate is known to Spring Security at all, and then will assign `GrantedAuthority` as per usual login rules.

Configuring Client Certificate authentication using the security namespace

With all the complexity of LDAP and OpenID configuration, configuring client certificate authentication is a welcome reprieve. If we are using the `security` namespace style of configuration, the addition of client certificate authentication is a simple one-line configuration change, added within the `<http>` declaration:

```
<http auto-config="true" ...>
<!-- Other content omitted -->
  <x509 user-service-ref="jdbcUserServiceCustom"/>
<!-- Other content omitted -->
</http>
```

After restarting the application, you'll again be prompted for a client certificate, but this time you should be able to access areas of the site requiring authorization. You can see from the logs (if you have them enabled) that you have been logged in as the `admin` user. Excellent job!

How Spring Security uses certificate information

As previously discussed, Spring Security's involvement in certificate exchange is to pick up information from the presented certificate, and mapping the user's credentials to a user service. What we did not see in the use of the `<x509>` declaration was the magic that makes this happen. Recall that when we had set up the client certificate, a distinguished name (DN) similar to an LDAP DN was associated with the certificate:

```
Owner: CN=admin, OU=JBCP Pets, O=JBCP Pets, L=Anywhere, ST=NH, C=US
```

Spring Security uses the information in this DN to determine the actual username of the principal and that it will look for in the `UserDetailsService`. In particular, it allows for specification of a regular expression, which is used to match a portion of the DN established with the certificate and utilize this portion of the DN as the principal name. The implicit, default configuration for the `<x509>` declaration would be as follows:

```
<x509
  subject-principal-regex="CN=(.*?),"
  user-service-ref="jdbcUserService"/>
```

We can see that this regular expression would match the value `admin` as the principal's name. This regular expression must contain a single matching group, but it can be configured to support the username and DN issuance requirements of your application—for example, if the DNs for your organization's certificates include the `email` or `userid` fields, the regular expression can be modified to use these values as the authenticated principal's name.

How Spring Security certificate authentication works

Let's review the various actors involved in the review and evaluation of the client certificates, and translation into a Spring Security authenticated session with the help of following diagram:

Anonymous User — Attempts to access Protected resource → `o.s.s.web.authentication.preauth.x509.X509AuthenticationFilter`

Continue Filter Chain →

Note some significant functionality is in `o.s.s.web.authentication.preauth.AbstractPreAuthenticatedProcessingFilter`

Is User Already Authenticated? — Yes

No

Located certificate in servlet request attributes? — No

Yes - extract principal from cert

`o.s.s.web.authentication.preauth.x509.X509PrincipalExtractor` — Implemented by → `o.s.s.web.authentication.preauth.x509.SubjectDnX509PrincipalExtractor`

Use `java.util.regex.Matcher` to locate principal through regex match

Regex matched certificate DN?

Yes → Build `o.s.s.web.authentication.preauth.PreAuthenticatedAuthenticationToken` for authentication

No → BadCredentialsException

We're back in `o.s.s.web.authentication.preauth.AbstractPreAuthenticatedProcessingFilter`

Attempt authentication with Authentication Manager reference

Successful?

No → Throw on failure? — Defaults to false

Yes → AuthenticationException

No

Yes → Set Authentication in security context

We can see that the `o.s.s.web.authentication.preauth.x509.`
`X509AuthenticationFilter` is responsible for examining the request of an
unauthenticated user for presentation of client certificates. If it sees the request
includes a valid client certificate, it will extract the principal using an `o.s.s.web.`
`authentication.preauth.x509.SubjectDnX509PrincipalExtractor`, using
regular expression matching on the certificate owner's DN, as previously described.

> Be aware that, although the diagram indicates that examination
> of the certificate occurs for unauthenticated users, a check is also
> done for presentment of a certificate for a different user than the
> one which was already used to authenticate the user. This would
> result in a new authentication request using the newly provided
> credentials. The reason for this should be clear—any time a user
> presents a new set of credentials, the application must be aware of
> this, and react in a responsible fashion by ensuring the user should
> still be able to access it.

Once the certificate has been accepted (or rejected/ignored), as with other
authentication mechanisms, an `Authentication` token is built, and passed
along to the `AuthenticationManager` for authentication. We can now review
the very brief illustration of the `o.s.s.web.authentication.preauth.`
`PreAuthenticatedAuthenticationProvider`'s handling of the authentication token:
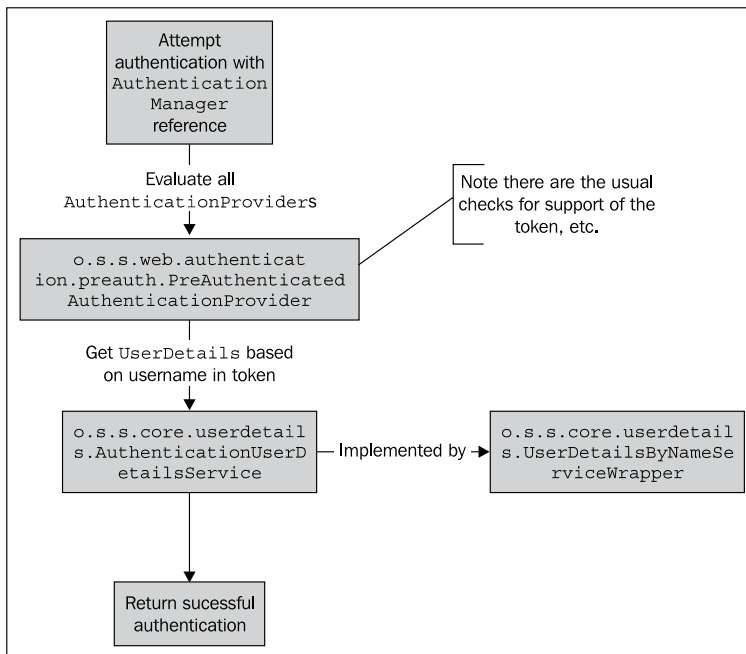
If you have read through our review of CAS authentication in *Chapter 10, Single Sign On with Central Authentication Service*, you may note some similarities between the processing of client certificate authenticated requests and CAS requests—this is intentional, and in fact a similar design affects other, less used, pre-authenticated mechanisms supported by Spring Security including Java EE role mapping and Site Minder-style authentication. If you understand the processing flow of client certificate authentication, understanding these other authentication types is significantly easier.

Other loose ends

One loose end that we need to tie up is the handling of authentication denial. In *Chapter 6, Advanced Configuration and Extension*, we learned about the functionality of the `AuthenticationEntryPoint` (we also revisited this topic in the CAS chapter). In a default form login scenario, the `LoginUrlAuthenticationEntryPoint` is used to redirect the user to a login page if they have been denied access to a protected resource and are not authenticated.

In contrast, in typical client certificate authentication environments, alternative methods of authentication are simply not supported (remember that Tomcat expects the certificate well before the Spring Security form login would take place anyway). As such, it doesn't make sense to retain the default behavior of redirection to a form login page. Instead, we'll modify the entry point to simply return an `HTTP 403 Forbidden` message, using the `o.s.s.web.authentication.Http403ForbiddenEntryPoint`. We'll configure the bean in `dogstore-base.xml`, where the rest of our Spring Beans are located as follows:

```
<bean id="forbiddenAuthEntryPoint" class="org.springframework.
security.web.authentication.Http403ForbiddenEntryPoint"/>
```

Next, a simple attribute assignment on the `<http>` declaration puts the new entry point in use immediately:

```
<http ... entry-point-ref="forbiddenAuthEntryPoint">
```

Now, if a user tries to access a protected resource and is unable to provide a valid certificate, they will be presented with the following page:
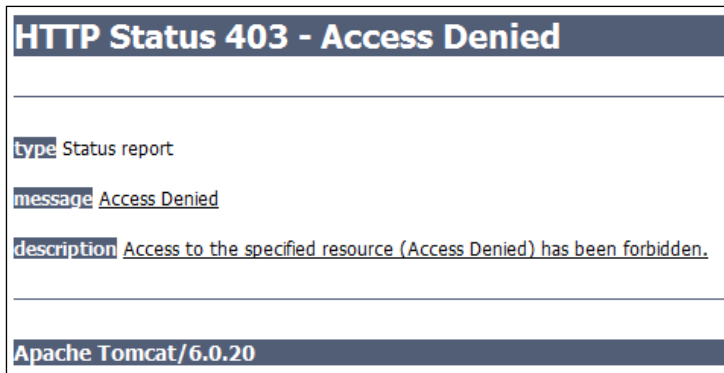
Note that unlike configuration of the `AccessDeniedHandler` that we saw in Chapter 6, the `Http403ForbiddenEntryPoint` cannot be reconfigured to redirect the user to a friendly page or Spring-managed URL. Instead, such configuration would need to be managed through the web application deployment descriptor's `<error-page>` declaration, as specified by the Java EE servlet specification, or by subclassing the `Http403ForbiddenEntryPoint` to override this behavior.

Other configuration or application flow adjustments that are commonly performed with client certificate authentication are as follows:

- Removal of the form-based login page altogether
- Removal of the **Log Out** link (as there's no reason to log out, because the browser will always present the user's certificate)
- Removal of functionality to rename the user account and change password.
- Removal of user registration functionality (unless you are able to tie it into the issuance of a new certificate)

Supporting Dual-Mode authentication

It is also possible that some environments may support both certificate-based and form-based authentication. If this is the case in your environment, it is also possible (and trivial) to support it with Spring Security 3. We can simply leave the default `AuthenticationEntryPoint` (redirecting to the form-based login page) intact and allow the user to log in using the standard login form, if they do not supply a client certificate.

If you choose to configure your application this way, you'll need to adjust the Tomcat SSL settings (change as appropriate for your application server). Simply change the `clientAuth` directive to `want` instead of `true`:

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true"
  maxThreads="150" scheme="https" secure="true"
  sslProtocol="TLS"
  keystoreFile="conf/tomcat.keystore"
  keystorePass="password"
  truststoreFile="conf/tomcat.truststore"
  truststorePass="password"
  clientAuth="want"
  />
```

We'll also need to remove the `entry-point-ref` that we configured in the previous exercise, so that the standard form-based authentication workflow takes over if the user isn't able to supply a valid certificate upon the browser first being queried.

Although this is convenient, there are a few things to keep in mind about dual-mode (form- and certificate-based) authentication.

> Most browsers will not re-prompt the user for a certificate if they have failed certificate authentication once, so make sure that your users are aware that they may need to re-enter the browser to present their certificate again.

Recall that a password is not required to authenticate users with certificates; however, if you are still using a JDBC `UserDetailsService` to support your form-based authenticated users, this may be the `UserDetailsService` that you are also using to give the `PreAuthenticatedAuthenticationProvider` information about your users. This presents a potential security risk, as users who you intend to sign in only with certificates could potentially authenticate using form login credentials. There are several ways to solve this problem, and they are described in the following list:

- Ensure that the users authenticating with certificates have an appropriately strong password in your JDBC user store
- Consider customizing your JDBC user store to clearly identify users who are enabled for form-based login. This can be tracked with an additional field on the table holding user account information, and minor adjustments to the SQL queries used by the `JdbcDaoImpl`
- Configure a separate user details store altogether for users who are logging in as certificate-authenticated users, to completely segregate them from users allowed to use form-based login

Dual-mode authentication can be a powerful addition to your site, and can be deployed effectively and securely, provided that you keep in mind the situations under which users will be granted access to it.

Configuring Client Certificate authentication using Spring Beans

Earlier in this chapter, we reviewed the flow of the classes involved in client certificate authentication. As such, it should be straightforward for us to configure JBCP Pets using the bean-only configuration from `dogstore-explicit-base.xml`. We'll add the following bean definitions, which correspond to all the beans we've discussed thus far:

```xml
<bean id="x509Filter" class="org.springframework.security.web.
authentication.preauth.x509.X509AuthenticationFilter">
  <property name="authenticationManager" ref="customAuthenticationMan
ager"/>
</bean>
<bean id="preauthAuthenticationProvider" class="org.
springframework.security.web.authentication.preauth.
PreAuthenticatedAuthenticationProvider">
  <property name="preAuthenticatedUserDetailsService" ref="authenticat
ionUserDetailsService"/>
</bean>
<bean id="forbiddenAuthEntryPoint" class="org.springframework.
security.web.authentication.Http403ForbiddenEntryPoint"/>
<bean id="authenticationUserDetailsService"
 class="org.springframework.security.core.userdetails.
UserDetailsByNameServiceWrapper">
  <property name="userDetailsService" ref="jdbcUserService"/>
</bean>
```

We'll also need to add the filter to our filter chain (and remove the login and logout related filters):

```xml
<bean id="springSecurityFilterChain" class="org.springframework.
security.web.FilterChainProxy">
  <security:filter-chain-map path-type="ant">
    <security:filter-chain pattern="/**" filters="
        securityContextPersistenceFilter,
        x509Filter,
        anonymousProcessingFilter,
        exceptionTranslationFilter,
        filterSecurityInterceptor" />
  </security:filter-chain-map>
</bean>
```

Finally, we'll need to add the `AuthenticationProvider` implementation to the `ProviderManager`, and remove all the others that are present:

```
<bean id="customAuthenticationManager" class="org.springframework.
security.authentication.ProviderManager">
  <property name="providers">
    <list>
      <ref local="preauthAuthenticationProvider"/>
    </list>
  </property>
</bean>
```

At this point, our bean-based configuration is ready for use. If you'd like to try it out, remember to switch the reference in `web.xml` to use the bean-only configuration mechanism!

Additional capabilities of bean-based configuration

The use of Spring bean-based configuration provides us with additional capabilities through exposure of bean properties, which aren't exposed through the security namespace style of configuration.

Additional properties available on the `X509AuthenticationFilter` are as follows:

Property	Description	Default
`continueFilterChainOn UnsuccessfulAuthentication`	If `false`, a failed authentication will throw an exception, rather than allowing the request to continue. This would typically be set in cases where a valid certificate is expected, and required, to access the secured site. If `true`, the filter chain will proceed, even if there is a failed authentication.	`true`
`checkForPrincipalChanges`	If `true`, the filter will check to see if the currently authenticated username differs from the username presented in the client certificate. If so, authentication against the new certificate will be performed, and the HTTP session will be invalidated (optionally, see the next attribute). If `false`, once the user is authenticated, they will remain authenticated even if they present different credentials.	`false`

Property	Description	Default
`invalidateSessionOn PrincipalChange`	If `true`, and the principal in the request changes, the user's HTTP session will be invalidated prior to being reauthenticated. If `false`, the session will remain—note that this may introduce security risks.	`true`

The `PreAuthenticatedAuthenticationProvider` has a couple of interesting properties available to us, which are listed in the following table:

Property	Description	Default
`preAuthenticated UserDetailsService`	Used to build a full `UserDetails` object from the username extracted from the certificate.	None
`throwExceptionWhen TokenRejected`	If `true`, a `BadCredentialsException` will be thrown if the token is not properly constructed (does not contain a username or certificate). Typically set to true in environments where certificates are used exclusively.	`false`

In addition to these properties, there are a number of other opportunities for implementing interfaces or extending classes involved in certificate authentication to further customize your implementation.

Considerations when implementing Client Certificate authentication

Client certificate authentication, while highly secure, isn't for everyone, and isn't appropriate for every situation.

Pros:

- Certificates establish a framework of mutual trust and verifiability that both parties (client and server) are who they say they are
- Certificate-based authentication, if implemented properly, is much more difficult to spoof or tamper with than other forms of authentication
- If a well-supported browser is used and configured correctly, client certificate authentication can effectively act as a single sign-on solution, enabling transparent login to all certificate-secured applications

Cons:

- Use of certificates typically requires the entire user population to have them. This can lead to both a user training burden, and an administrative burden. Most organizations deploying certificate-based authentication on a large scale must have sufficient self-service and helpdesk support for certificate maintenance, expiration tracking, and user assistance
- Use of certificates is generally an all-or-none affair, meaning that mixed-mode authentication, offering support for non-certificate users, is not provided due to complexity of web server configuration or poor application support
- Use of certificates may not be well supported by all users in your user population, including mobile devices
- Correct configuration of the infrastructure required to support certificate-based authentication may require advanced IT knowledge

As we can see, there are both benefits and drawbacks to client certificate authentication. When implemented correctly, it can be a very convenient mode of access for your users, and has extremely attractive security and non-repudiation properties. You will need to determine for your particular situation whether or not this type of authentication is appropriate.

Summary

In this chapter, we examined the architecture, flow, and Spring Security support for client certificate-based authentication. We have covered the following:

- Reviewed the concepts and overall flow of client certificate (mutual) authentication
- Learned the important steps required to configure Apache Tomcat for a self-signed SSL and client certificate scenario
- Configured Spring Security to understand certificate-based credentials presented by clients
- Understood the architecture of Spring Security classes related to certificate authentication
- Discovered how to configure a Spring bean-style client certificate environment
- Weighed the pros and cons of this type of authentication

It's quite common for developers unfamiliar with client certificates to be confused by many of the complexities of this type of environment. We hope that this chapter has made this complicated subject a bit easier to understand and implement!

12
Spring Security Extensions

In this chapter, we'll explore the capabilities of one of the Spring Security Extensions project implementations—an exciting capability to integrate Windows Active Directory authentication (or other Kerberos-enabled infrastructure) with Spring Security to provide a seamless single-sign on experience for your intranet users.

During the course of this chapter, we will:

- Learn about the Kerberos authentication protocol and its adaptation to web-based credentials presentation
- Understand how a Kerberos-enabled web application fits into a Kerberos-based secure infrastructure
- Configure the JBCP Pets application to provide single sign-on authentication for Windows users, using an Active Directory authentication store
- Explore methods of using Active Directory as an LDAP `UserDetailsService`, to provide you with a single point of reference for user data

Spring Security Extensions

The **Spring Security Extensions** project (available at `http://static. springsource.org/spring-security/site/extensions.html`) functions as an incubator for useful extensions to Spring Security that are built upon the core Spring Security framework. Although the project is relatively new, it already has three exciting modules covering Kerberos authentication (the successor to NTLM authentication in Spring Security 2), Security Assertion Markup Language 2.0 authentication, and Portlet authentication.

In this chapter, we will cover the basic configuration and usage of Kerberos SPNEGO authentication. At the time of this writing, none of the Spring Security Extensions projects were officially released, but the Kerberos project is far enough along that the configuration shouldn't change significantly by the time you read this. Consider this chapter a peek into the experimental side of Spring Security through these unofficial, community-developed extensions.

A primer on Kerberos and SPNEGO authentication

Kerberos is a mutual authentication protocol used for authenticating clients—either individual users or network resources—against a centralized credentials repository known as the **key distribution center** (**KDC**). The negotiation between the client and KDC is quite involved, and well documented in several internet standards (primarily RFC 4120, The Kerberos Network Authentication Service (V5), available at `http://tools.ietf.org/html/rfc4120`).

For the purposes of the discussion in this chapter, we will simplify the level of detail with which we describe the credential checking activity as it relates to the Kerberos infrastructure. The general purpose of Kerberos authentication and Kerberos infrastructure is to provide secure and trustworthy authentication of principals, whether they be individual users, network resources, or software applications. Kerberos describes both the security protocol itself and a software implementation originally developed at the Massachusetts Institute of Technology (MIT). Due to the fact that Kerberos has such a robust history, there are many excellent books covering Kerberos in a high level of detail.

On top of the Kerberos authentication protocol, the **Generic Security Service Application Program Interface** (**GSS-API**) was developed, also through an RFC standard (RFC 2078, Generic Security Service Application Program Interface, Version 2, `http://tools.ietf.org/html/rfc2078`). The GSS-API provides a standard, cross-platform, cross-language authentication and security API, wrapping Kerberos (and other authentication providers). Its development goal was to provide security developers with a consistent high-level programming interface for authentication and security without needing to know the details of a particular security protocol.

A standard implementation of the GSS-API in the Java language is defined in another RFC standard (RFC 2853, Generic Security Service API Version 2: Java Bindings, `http://tools.ietf.org/html/rfc2853`), which is implemented in the Sun JVM in the `org.ietf.jgss` package.

> An important thing to understand about Kerberos and GSS-API
> support in Java is that the code to implement this API, in addition
> to the API itself, is part of the JRE and does not require the inclusion
> of any third-party libraries. In fact, you may consult the Sun website
> (`http://java.sun.com/javase/6/docs/technotes/guides/`
> `security/jgss/tutorials/index.html`) for some significant
> documentation of these capabilities of the Sun JVM. Be aware that
> other JVM implementations may not have the same capabilities as the
> Sun JVM in this area.

The wiring of GSS-API authentication to a web browser occurs through the use of a
message exchange standard known as **Simple and Protected GSS-API Negotiation
Mechanism** (**SPNEGO**). SPNEGO is an algorithm specifying the behavior of two
GSS-API implementations in determining whether they can communicate with each
other over a common security protocol. In the case of Kerberos SPNEGO, we hope
that the common security protocol through which client and server negotiate
is Kerberos.

Although SPNEGO can be used when communicating between GSS-API-enabled
applications or systems, it is the adaptation of SPNEGO for use in web client
authentication that has cemented this technology in the lexicon of web application
developers and security designers. The Microsoft-sponsored RFC 4559
(SPNEGO-based Kerberos and NTLM HTTP Authentication in Microsoft
Windows, `http://tools.ietf.org/html/rfc4559`) describes behavior by which
a server can request GSS-API authentication using an SPNEGO request sent over
the HTTP protocol. The client browser can then respond to the SPNEGO request
and collect credentials from the end user using the native operating system's
GSS-API support. Microsoft developed this specification to support two security
implementations of GSS-API for its Windows operating system—its proprietary
NT LAN Manager (**NTLM**) protocol and the open Kerberos protocol. Eventually,
the convenience of single sign-on from Windows-based clients to Kerberos-enabled
server resources led to the adaptation of browser-based SPNEGO in other browsers,
including Mozilla Firefox and Apple Safari.

Hopefully we didn't lose you in terminology soup! Let's get real and review how this
all works in a web-based authentication scenario.

The following diagram illustrates the interaction between the three participants in the SPNEGO Kerberos authentication process:
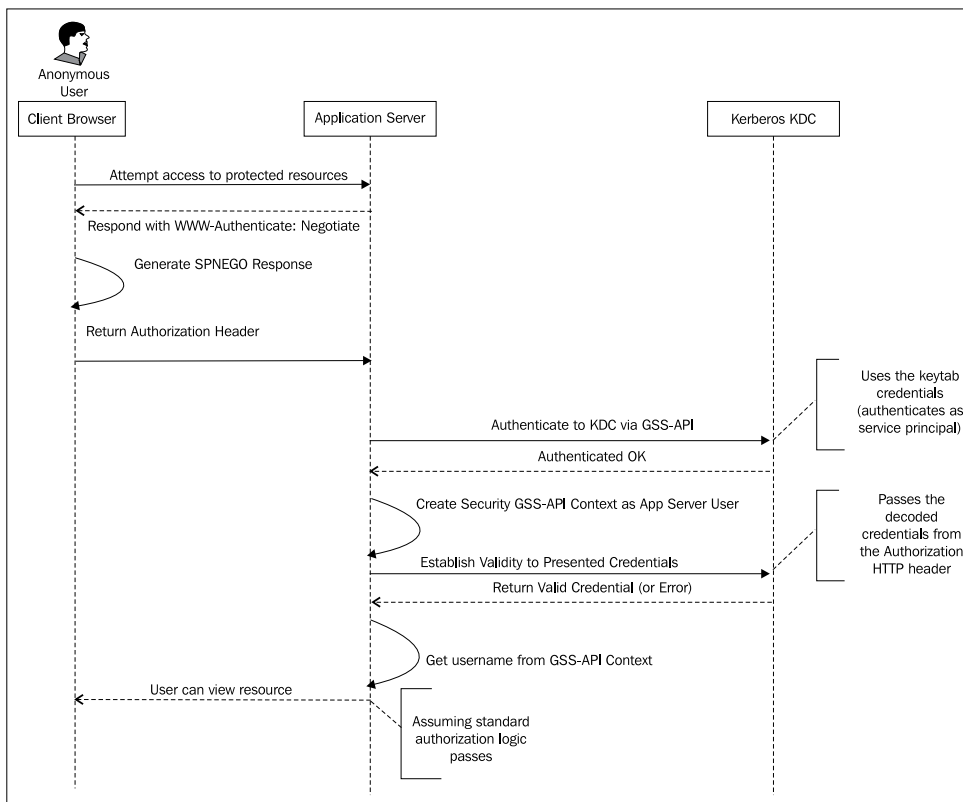
There are two critical pieces in the HTTP exchange between the client browser and the application server that are important to understand before jumping into the Spring Security implementation of SPNEGO authentication via Kerberos.

- When authentication is required, the application server must send an HTTP response header with the value `WWW-Authenticate: Negotiate` to the client. This tells the client that authentication is required, using the SPNEGO protocol.

- The client will determine the user's credentials (which may result in a prompt, depending on the browser implementation and operating system), and respond to the server with the credentials encoded in the HTTP `Authorize` request header.

Here we focus more on the browser to application server interaction, because this is the portion that most concerns Spring Security integrators.
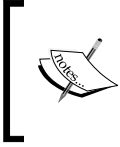
> Although it is not required for successful implementation of SPNEGO, we recommend the use of SSL in a production environment, so that SSO credentials remain secure throughout the web client authentication lifecycle.

Another important piece of infrastructure to consider is the Kerberos KDC implementation. Open source options exist, most notably, the Kerberos implementation from MIT (the famous university), which was one of the original sponsors of the Kerberos technology; however, the most common implementation developers are likely to see in an enterprise environment is Microsoft **Active Directory** (AD). Windows Server implementations of Active Directory can serve a dual function as Kerberos KDCs, and as such if an organization is using AD, it is automatically enabled for Kerberos-based authentication as well.

Kerberos authentication in Spring Security

Much as we saw with CAS and client certificate authentication, it is the intention that Kerberos **Single Sign-On** (SSO) be the only supported authentication mechanism for the Kerberos secured website (although we will see an alternative configuration supporting form login towards the end of this chapter). Let's review the basic steps of configuring Kerberos authentication for Spring Security; afterwards, we'll cover some of the required configuration steps for other participants in the Kerberos-secured environment.

Overall Kerberos Spring Security authentication flow

As we discussed in the overview of the SPNEGO protocol design, the key piece of the implementation is the exchange of messages between the client's browser and the secured application. This is where Spring Security's Kerberos Extension comes in—to handle the negotiation of authentication credentials.

As with a typical external authentication store, such as we've seen with CAS (in *Chapter 10*, *Single Sign On with Central Authentication Service* and in *Chapter 11*, *Client Certificate Authentication*), it is the combined responsibility of a servlet filter and a custom `AuthenticationProvider` to perform the authentication with the external store and verify that the returned response is trustworthy and verifiable.

Let's review the important classes involved in the process of SPNEGO authentication with Spring Security using the following diagram:
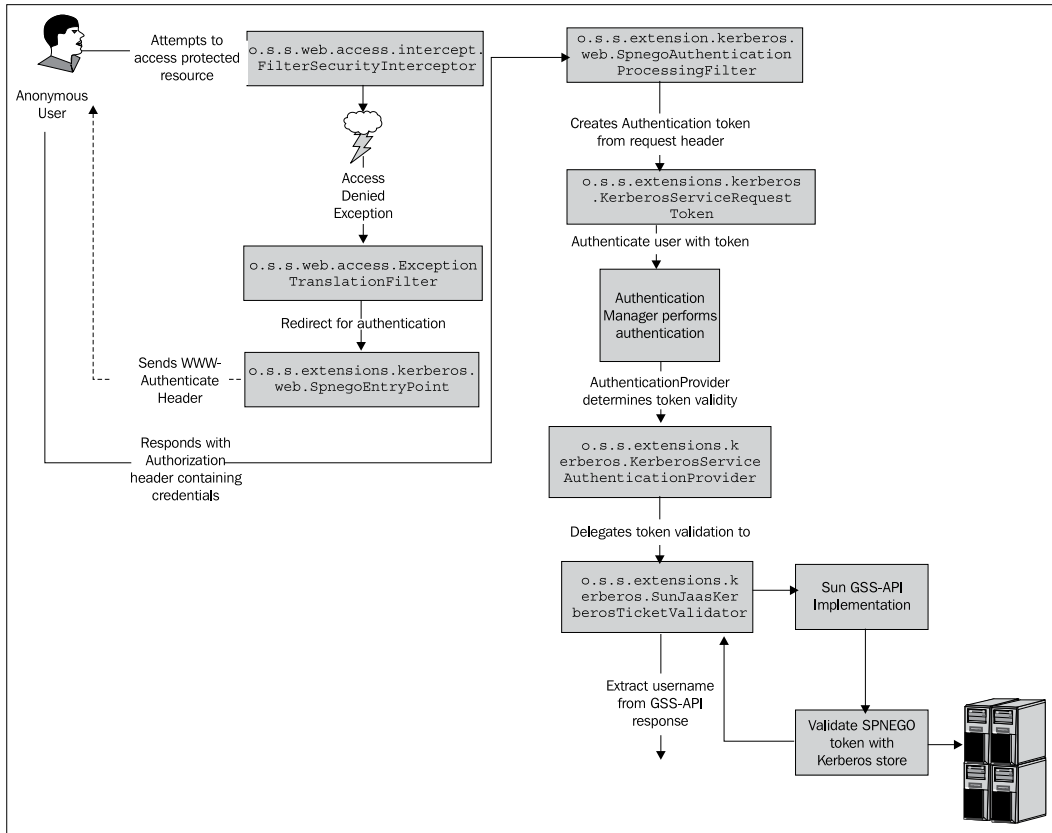
We can see how the services of the `o.s.s.extensions.kerberos.`
`KerberosServiceAuthenticationProvider` are primarily responsible for coordinating the validation of the SPNEGO response from the browser. In just a moment, we'll dive into the details of the Spring Bean configuration required to wire these classes together!

Getting prepared

The configuration of any Kerberos environment, especially if it is backed by Microsoft Active Directory, can be time consuming and complicated, especially if you do not have prior experience with Kerberos. If you are deploying a Kerberos-enabled web application into an environment, make sure the environment is correctly configured before you start, otherwise you may spend more time diagnosing the environment than developing your application.

We'll assume for the purposes of this tutorial that you are writing the application to authenticate in a Microsoft Active Directory (AD) environment, which is (for most users) the most likely scenario to combine Spring Security with Kerberos. If this describes you, when you read "Kerberos principal", simply think of this as being equivalent to an AD user.

Before you start, make sure:

- You have set up a Kerberos principal for the web application itself. We will perform additional required configuration on the application server using this account.
- The computer from which you are connecting to the website (the one on which your web browser is running) must be part of the Kerberos authentication realm. For Windows users, this means that the computer must be part of the AD domain.
- The computer running the web browser must be different than the computer running the web application. A virtual machine is handy for this.

Remember that Kerberos SSO is typically deployed for intranet applications running within an AD or Kerberos Realm, and wouldn't be used for public-facing web applications.

Assumptions for our examples

We'll assume the following setup for the Kerberos examples in this chapter:

Configuration	Value	Description
Domain name	`jbcppets.com`	The domain name of our corporate network and website.
AD domain	`corp.jbcppets.com`	The Active Directory domain—matches the Kerberos realm name (abbreviated as `CORP` for clarity)
Website principal	`CORP\website`	The AD user corresponding to the website service principal.

Creating a keytab file

You will need to create a keytab file, which will be used to authenticate the web application to the KDC and allow further authentication by proxy for credentials presented by the user. A **keytab** file is an encrypted copy of the private key for a Kerberos principal, and can function in lieu of a password for authenticating a principal to the Kerberos server.

Our application is using web-based SPNEGO Kerberos authentication. The relevant protocol RFC (RFC 4559) specifies that the keytab must map to a principal with the identifier `HTTP/fully.qualified.web.server.name` (this must be exactly as specified). In our sample configuration, we will assume that the JBCP Pets web application is running at `web.jbcppets.com` within the Kerberos domain `corp.jbcppets.com`, so the principal name expected in the keytab would be `HTTP/web.jbcppets.com@CORP.JBCPPETS.COM`.

As we are running AD as our Kerberos KDC, we will need to map the AD user `CORP\website` to the required Kerberos principal. We can do this using the Microsoft `ktpass` tool (on the AD domain controller) as follows:

```
ktpass -princ HTTP/web.jbcppets.com@CORP.JBCPPETS.COM -mapuser CORP\
website -out website.keytab
```

You should be able to visually map the arguments of this command to the configuration scenario that we've described as follows:

- `HTTP/web.jbcppets.com@CORP.JBCPPETS.COM`: The name of the Kerberos principal (`HTTP/web.jbcppets.com`) and realm (`CORP.JBCPPETS.COM`)
- `CORP\website`: The name of the domain user to map to the Kerberos principal

Take note that Kerberos realms are case sensitive, so make sure that you are consistent with case across all aspects of your environment. By convention, Kerberos realms are usually specified in uppercase.

This will create a file called `website.keytab` in the current directory. You'll need to securely transfer this file to the machine running the web application in order to configure Kerberos within Spring Security. Place it in the `WEB-INF/classes` directory of the JBCP Pets web application—it'll be referenced in just a moment when we configure the Spring Security Kerberos beans.

> Be very careful with the keytab file—it contains information equivalent to a pre-authenticated set of credentials to your Kerberos realm. Do not use insecure file transfer techniques to copy this file from the source to destination servers, because this would leave you vulnerable to a network sniffer. If the files were compromised, a malicious user could use these credentials to log into your Kerberos realm as the web application user!

Note that `ktpass` is included with Windows 2008 Server or later. Earlier versions of Windows Server may need to install Kerberos support files to be able to use these command line tools.

Configuring Kerberos-related Spring beans

As most of the Kerberos authentication takes place within the Sun JVM itself, there's not a lot of configuration (or even code) behind Spring Security Kerberos authentication.

First, we'll need to configure the `AuthenticationEntryPoint` responsible for sending the `WWW-Authenticate` header that is the first step in the client machine responding with Kerberos credentials. The following bean declaration in `dogstore-base.xml` will be as follows:

```
<bean id="kerbEntryPoint" class="org.springframework.security.
extensions.kerberos.web.SpnegoEntryPoint" />
```

Next, we'll need to define the filter that will parse the incoming `Authorization` HTTP request header and process this as a SPNEGO sign-on request as follows:

```
<bean id="kerbAuthenticationProcessingFilter"
 class="org.springframework.security.extensions.kerberos.web.
SpnegoAuthenticationProcessingFilter">
  <property name="authenticationManager" ref="authenticationManager"
/>
</bean>
```

Remember that, as with other SSO implementations we've seen to this point, this filter is responsible for both parsing the expected SSO header and actually attempting to authenticate the user based on that header. The `SpnegoAuthenticationProcessingFilter` will populate an `o.s.s.extensions. kerberos.KerberosServiceRequestToken` with the credentials found in the HTTP `Authorization` header.

Now we need an `AuthenticationProvider` that will take the credentials presented in the populated `KerberosServiceRequestToken` and validate them. Similar to what we saw in CAS authentication, it's the responsibility of the Kerberos `AuthenticationProvider` to validate the token against the ticket granter.

```
<bean id="kerberosServiceAuthenticationProvider"
 class="org.springframework.security.extensions.kerberos.
KerberosServiceAuthenticationProvider">
  <property name="ticketValidator" ref="ticketValidator"/>
  <property name="userDetailsService" ref="jdbcUserService" />
</bean>
<bean id="ticketValidator" class="org.springframework.security.
extensions.kerberos.SunJaasKerberosTicketValidator">
  <property name="servicePrincipal" value="HTTP/web.jbcppets.com@CORP.
JBCPPETS.COM" />
  <property name="keyTabLocation" value="classpath:website.keytab"/>
</bean>
```

The `KerberosServiceAuthenticationProvider` requires a reference to a delegate `o.s.s.extensions.kerberos.KerberosTicketValidator` implementation, which is used to validate the Kerberos ticket obtained from the authentication token. The only implementation provided by the Spring Security Kerberos Extension is an implementation that utilizes the Sun JVM's GSS-API to both validate the contents of the keytab (that they match the principal configured in the `servicePrincipal` attribute) and perform validation of the Kerberos ticket on behalf of the service principal.

We recognize that the `servicePrincipal` and `keyTabLocation` are references to the configurations that we made earlier on the Kerberos server.

Where should the keytab file be placed?

Recall that the keytab file effectively presents a high risk security element, and as such, it's not recommended that you put it on the application classpath (although this is certainly convenient for development). Instead, it's recommended that you place this on the file system outside of your web application deployment directory, in a secure location. You may reference the file using the Spring standard `file:` syntax.

You can see that we've configured (for the moment) a reference to the JDBC `UserDetailsService`. You'll have to remember to add a new user to the JDBC bootstrap SQL with a username matching the Kerberos identity of the user you're going to try to log in with. For example, we'll try logging in to the application with the user `kerbuser`, so the full username in the database should be the fully-qualified Kerberos principal `kerbuser@CORP.JBCPPETS.COM`. Add this to `WEB-INF/classes/test-users-groups-data.sql` as follows:

```
insert into users(username, password, enabled, salt) values
('kerbuser@CORP.JBCPPETS.COM','unused',true,CAST(RAND()*1000000000 AS
varchar));
insert into group_members(group_id, username) select id,'kerbuser@
CORP.JBCPPETS.COM' from groups where group_name='Administrators';
```

Keep in mind that with Active Directory, it makes a lot more sense to use an LDAP `UserDetailsService`, or even a custom one specifically suited to your business needs. We'll cover this style of configuration in a moment.

Wiring SPNEGO beans to the security namespace

We will now need to wire up the beans that we've configured to the `security` namespace configuration elements. First, we'll need to add a reference to our new `AuthenticationEntryPoint` (much as we did with CAS configuration in Chapter 10) as follows:

```
<http ...
  entry-point-ref="kerbEntryPoint">
```

Now we'll insert the SPNEGO filter into the Spring Security filter chain. A suitable location would be to replace the form login filter with the SPNEGO one, so let's do that:

```
<custom-filter ref="kerbAuthenticationProcessingFilter"
position="FORM_LOGIN_FILTER" />
```

Finally, we'll need an `<authentication-provider>` reference to the new `AuthenticationProvider` responsible for handling SPNEGO tickets. Let's add that now:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider ref="kerberosServiceAuthenticationProvider
"/>
</authentication-manager>
```

That's all the additional configuration required to enable SPNEGO SSO security for our application!

If your machine is already properly configured for Kerberos authentication, you may be able to start the web application now and try to authenticate to it from another machine that is on the domain.

Try to use Internet Explorer for the first test; IE requires little to no configuration in order to allow the user to respond manually to the SPNEGO request. You should see a prompt similar to the following as soon as you hit a protected page such as the **My Account** page:
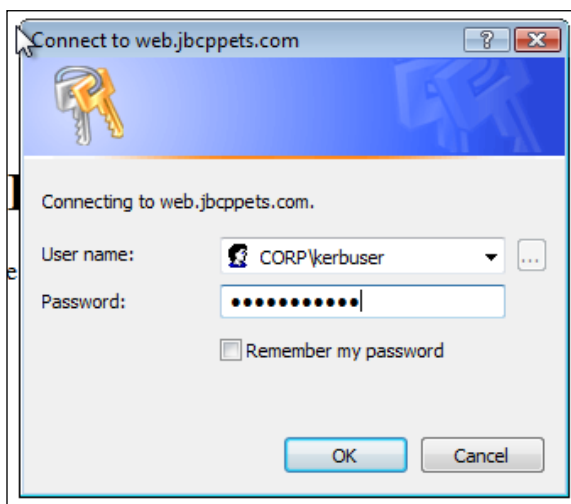
Congratulations—if you see this prompt, and then are able to view the **My Account** page after presenting the correct credentials, you have successfully configured Kerberos authentication for your web application!

Next up is to configure Internet Explorer to pass the user's network credentials upon demand. The following configuration settings must be present in order for IE to respond transparently to the SPNEGO SSO request:

- Ensure that the site is added to the **Local Intranet** security zone. You can manually add your site to the security zone using the **Security** tab of the **Internet Options** dialog for Internet Explorer.

- Ensure that Internet Explorer Protected Mode is enabled for the intranet zone by checking the **Enable Protected Mode** checkbox.

- On the **Advanced** tab, ensure that the **Enable Integrated Windows Authentication** checkbox is checked.

With all of these settings successfully verified (and Internet Explorer restarted), you should see that the credentials of the logged-in domain user are automatically presented to the site via SPNEGO authentication, and the user is immediately logged in.

Adding the Application Server machine to a Kerberos realm

One final step if you are working on a new machine—to configure the system-wide Kerberos settings so that the Sun GSS-API can determine where to find the Kerberos KDC for the realm (or realms) to which users will be authenticating. This requires the creation of a `krb5.ini` file, which is typically located in the Windows installation directory (`c:\Windows`, or equivalent).

While the configuration syntax of this file is outside of the scope of this book, the following basic configuration is sufficient for a single-realm environment, with the KDC located at `corp.jbcppets.com`.

```
[domain_realm]
.jbcppets.com = CORP.JBCPPETS.COM
[libdefaults]
default_realm = CORP.JBCPPETS.COM
[logging]
[realms]
CORP.JBCPPETS.COM = {

  kdc = corp.jbcppets.com

}
```

Instead of using the `krb5.ini` file, it is also possible to specify the default realm and KDC using the JVM properties indicated in the following table:

Parameter	Description	Example
`-Djava.security.krb5.realm`	Default realm	`CORP.JBCPPETS.COM`
`-Djava.security.krb5.kdc`	Default KDC	`corp.jbcppets.com`

Generally speaking, we would recommend use of the `krb5.ini` file, as this file may also be used by other Kerberos-enabled applications.

Special considerations for Firefox users

By default, Firefox does not support Kerberos authentication. This has been an historical oddity (there are claims by some that this increases security), but regardless, out of the box, Firefox simply will not prompt users when presented with a `WWW-Authenticate: Negotiate` request header.

Firefox users must modify a setting in the browser's `about:config` screen to explicitly add the domain (or domains) for which credentials will automatically be sent, by providing the domain name in the `network.negotiate-auth.trusted-uris` parameter, as illustrated in the following screenshot:
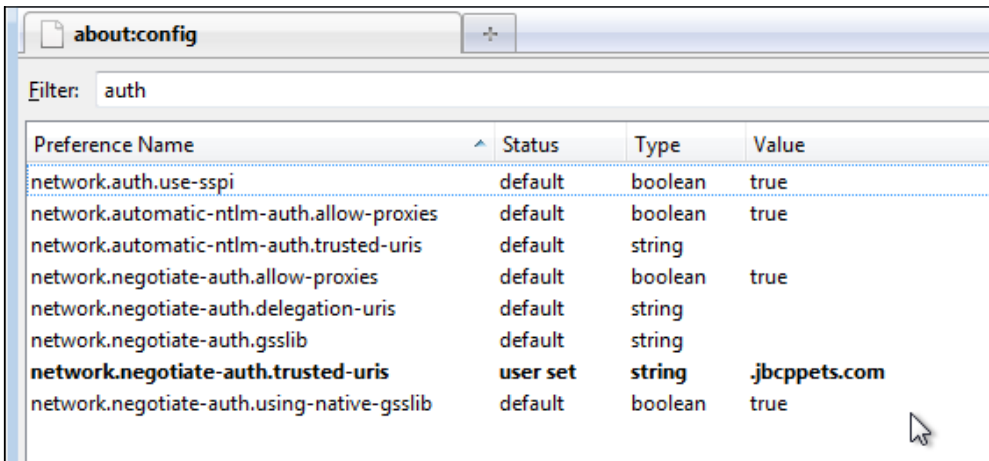
With this setting altered from the default (empty) setting, Firefox should automatically pass your Windows domain credentials to the site upon request. It's unfortunate that this option is hidden from casual users, as if this were changed Firefox would be much easier to configure for SPNEGO than Internet Explorer!

Troubleshooting

Unfortunately, Kerberos can be extremely complicated to configure correctly (especially for first-time developers or administrators), and Kerberos-enabling web applications adds to this complexity.

Verifying connectivity with standard tools

First and foremost, make sure that the underlying Kerberos infrastructure works properly. This means that you should be able to verify, using standard Kerberos tools (such as `kinit` or `ktab`), or a graphical Kerberos client such as the MIT Kerberos for Windows (KfW) client (available at `http://web.mit.edu/Kerberos/`).

`kinit` and `ktab` are part of a standard JDK distribution—a sample command line for `ktab` is illustrated as follows:

```
ktab -a kerbuser@CORP.JBCPPETS.COM -k kerbuser.keytab
Password for kerbuser@CORP.JBCPPETS.COM:xxxxx
Done!
Service key for kerbuser@CORP.JBCPPETS.COM is saved in kerbuser.keytab
```

The MIT Kerberos implementation provides a graphical login and configuration interface, and may be easier for new users to grasp.

Enabling Java GSS-API debugging

There's not much logging in the Java GSS-API, and as this library does the bulk of the work in authenticating via Kerberos, it's helpful to be able to get a bit of a view into what it does. You may either set the JVM property `-Dsun.security.krb5.debug=true` or declare the `debug` property of the `SunJaasKerberosTicketValidator` bean as follows:

```xml
<bean id="ticketValidator" class="org.springframework.security.
extensions.kerberos.SunJaasKerberosTicketValidator">
   <property name="servicePrincipal" value="HTTP/web.jbcppets.com@corp.
jbcppets.com" />
    <property name="keyTabLocation" value="classpath:website.keytab"/>
   <property name="debug" value="true"/>
</bean>
```

Either of these should output a bit more helpful information to the console of the application server in the case of failure.

Other troubleshooting steps

A few more general suggestions wrap up the list of common issues when
kerberizing (applying Kerberos to) your web application with the
Spring Security Kerberos Extension:

- Make sure that you have standard logging for Spring Security, at least
  WARN level or higher, enabled for the `org.springframework.security.`
  `extensions.kerberos` package. If configuration problems or ticket
  validation problems exist on the application server, these are commonly
  surfaced in standard Spring logging.

- Use a network traffic monitor or packet sniffer to watch the traffic
  between the application server and the KDC. This type of tool will also
  help when monitoring the exchange between the client browser and the
  application server.

- If you are using Windows, do not attempt to use the browser on the same
  machine as the application server to conduct testing—it will not work! This
  is because Windows will automatically use NTLM authentication. NTLM
  authentication uses a similar request/response header exchange, but with
  different data formats. A way to identify this problem is to look in the Spring
  logs for the Negotiate header returned by the client browser. If the header
  starts with TlRM, it is an NTLM header and not SPNEGO. SPNEGO headers
  will start with YII and should be quite long.

- Make sure you are logged in to Windows with a domain account
  and not a local machine account, otherwise Windows will attempt
  NTLM authentication.

- Make sure that DNS resolution for all participants in the Kerberos
  authentication process works from both the server and client machines.

Even with these troubleshooting steps, you may find that successful deployment of
SPNEGO and Kerberos is quite challenging—don't give up, you can get it working!

Configuring LDAP UserDetailsService with Kerberos

For convenience, we have configured the JDBC `UserDetailsService` and hardcoded the Kerberos user IDs. Usually when configuring Kerberos authentication (especially with Active Directory), the user details will be retrieved from AD via LDAP. We'll walk through a typical configuration of an LDAP `UserDetailsService` against Microsoft AD when using Kerberos authentication. We'll skip much of the explanation of Spring Security's integration with LDAP, although you're welcome to jump back to *Chapter 9*, *LDAP Directory Services* where we covered LDAP in great detail, including some specific suggestions about integration with AD via LDAP.

We can configure the authentication provider to reference an LDAP `UserDetailsService` that can be defined in the `dogstore-security.xml` file as follows:

```
<ldap-server url="ldaps://corp.jbcppets.com/DC=corp,DC=jbcppets,DC=com"
  id="ldapCorp" manager-dn="CN=Administrator,CN=Users,DC=corp,DC=jbcppets,DC=com"
  manager-password="apassword!"/>
<ldap-user-service id="ldapUserService" server-ref="ldapCorp"
  user-search-filter="(userPrincipalName={0})" user-search-base="CN=Users"
  group-search-base="CN=Groups"/>
```

The key here is the `user-search-filter`, which searches by the `userPrincipalName` LDAP attribute—this matches the Kerberos principal name provided during SPNEGO authentication. Take care that you provide a `manager-dn`, which simply has access to the AD and is not actually a domain administrator!

Next, simply adjust the `KerberosServiceAuthenticationProvider` to reference this `UserDetailsService` in `dogstore-base.xml` as follows:

```
<bean id="kerberosServiceAuthenticationProvider"
  class="org.springframework.security.extensions.kerberos.
KerberosServiceAuthenticationProvider">
  <property name="ticketValidator" ref="ticketValidator"/>
  <property name="userDetailsService" ref="ldapUserService" />
</bean>
```

The only caveat with this configuration technique is the mapping of authorities to the authenticated user. Recall from Chapter 9 that Spring Security LDAP expects the user authorities to be retrievable from LDAP in a particular way—if your AD is not set up in a way that is compatible with the out of the box LDAP user mapping, you may be required to write your own `LdapAuthoritiesPopulator` implementation. Should you need guidance in this area, please consult the implementation classes for this interface for direction and a good starting point.

Also note that Active Directory itself can be extremely complex and difficult to map directly in all cases. You may need to have conversations with the IT personnel who manage the Active Directory that you are attempting to integrate with, in order to determine the business requirements and appropriate application model to fit the needs of your users, and the needs of the business.

Using form login with Kerberos

Although use of SPNEGO authentication for browser-based SSO is a very common motivation for integrating Kerberos with Spring Security, it may be that you want to simply use Kerberos login itself as an `AuthenticationProvider` mechanism (similar to the way that LDAP bind authentication works—refer to Chapter 9 for details on this).

The benefit of configuring Kerberos authentication in this way is that we can seamlessly support Kerberos authenticated users alongside users authenticated using other `AuthenticationProviders` (LDAP, JDBC, and so on.).
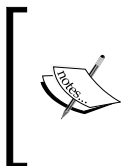
> If you are trying out this exercise, make sure to remove the configurations we performed in the first section of this chapter – most importantly, the `SpnegoEntryPoint` must be removed so that the user is redirected to the login form, otherwise users without SPNEGO will simply be denied access.

Let's now review the configuration steps. The Spring Beans that must be configured in `dogstore-base.xml` really don't overlap at all with the ones that we've configured for SPNEGO style login, so you're welcome to configure both at the same time (although realistically, as you can't easily have SPNEGO and form login simultaneously, you would probably not do this).

The following beans are required:

```
<bean id="kerberosAuthenticationProvider" class="org.springframework.
security.extensions.kerberos.KerberosAuthenticationProvider">
  <property name="kerberosClient" ref="kerbJaasClient"/>
  <property name="userDetailsService" ref="jdbcUserServiceCustom"/>
</bean>
<bean id="kerbJaasClient" class="org.springframework.security.
extensions.kerberos.SunJaasKerberosClient">
  <property name="debug" value="true"/>
</bean>
<bean id="kerbGlobalJaasConfig" class="org.springframework.security.
extensions.kerberos.GlobalSunJaasKerberosConfig">
  <property name="debug" value="true"/>
  <property name="krbConfLocation" value="/path/to/krb5.conf" />
</bean>
```

One aspect of this configuration that we've highlighted above is the
`krbConfLocation` property, which points to a Kerberos V5 configuration file. This
file can be formatted in exactly the same way as the sample `krb5.ini` file provided
earlier in the chapter (if you're curious about the meaning of the contents of the file,
please consult the relevant Kerberos V5 documentation page at `http://web.mit.edu/kerberos/krb5-1.5/krb5-1.5.1/doc/krb5-admin/krb5.conf.html`).

> Please take note that, unlike the earlier references to the Kerberos keytab
> file, the value of this parameter must be a physical, canonical pathname on
> the file system (for example, `c:\spring\krb5.conf`), and not a Spring-
> style `file:` or `classpath:` reference. This is because the file path is
> passed directly to the Sun JVM Kerberos subsystem, and not interpreted by
> a Spring Bean, as the `SunJaasKerberosTicketValidator` does.
>
> Also, take note that this style of configuration changes JVM property
> settings, so it may affect other Kerberos-enabled applications deployed in
> the same JVM.

The `default_realm` listed in the configuration file is utilized to authenticate user
credentials provided without a domain name through the user interface login form.
Users can also provide credentials with a domain name (for example, `kerbuser@jbcppets.com`), and normal Kerberos KDC resolution rules will apply based on
the Kerberos configuration.

As the intention of this style of Kerberos configuration is to support form-based login, we simply need to configure the `KerberosAuthenticationProvider` as we would configure for any other `AuthenticationProvider` in the `security` namespace configuration file `dogstore-security.xml` as follows:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider ref="kerberosAuthenticationProvider"/>
</authentication-manager>
```

After these configuration adjustments, you should be able to restart the application and use the form login to log in to your Kerberos back-end. Remember that you'll still need a `UserDetailsService` implementation to resolve `GrantedAuthority` mappings for users. Keep in mind that you can use this type of authentication for other `AuthenticationProvider`-backed authentication techniques, including basic authentication.

Summary

In this chapter, we learned how to adapt a Kerberos environment, such as that offered by Microsoft Active Directory within a Windows Domain, to supply integrated single sign-on for users of Windows operating systems. This provides a seamless, high-quality login experience to users, and eases the burden on systems administrators. In this chapter we have:

- Studied an overview of the important elements behind the Kerberos SPNEGO authentication protocol
- Learned about the infrastructure requirements and important setup steps to support a Kerberos-enabled web application
- Configured JBCP Pets to support SPNEGO Kerberos-backed single sign-on authentication
- Reviewed common troubleshooting steps for Kerberos-enabled web applications
- Examined the configuration required to expand to use AD as an LDAP backing store for user information
- Learned how to configure form-based login with a Kerberos back-end system

The next and final chapter will cover topics regarding migration from earlier versions of Spring Security. We hope you will enjoy it!

13
Migration to Spring Security 3

In this final chapter, we will review information relating to common migration issues when moving from Spring Security 2 to Spring Security 3.

During the course of this chapter we'll:

- Review important enhancements in Spring Security 3
- Understand configuration changes required in your existing Spring Security 2 applications when moving them to Spring Security 3
- Illustrate the overall movement of important classes and packages in Spring Security 3

Once you have completed the review of this chapter, you will be in a good position to migrate an existing application from Spring Security 2 to Spring Security 3.

Migrating from Spring Security 2

You may be planning to migrate an existing application to Spring Security 3, or trying to add functionality to a Spring Security 2 application and looking for guidance in the pages of this book. We'll try to address both of your concerns in this chapter.

First, we'll run through the important differences between Spring Security 2 and 3—both in terms of features and configuration. Second, we'll provide some guidance in mapping configuration or class name changes. These will better enable you to translate the examples in the book from Spring Security 3 back to Spring Security 2 (where applicable).

A very important migration note is that Spring Security 3 mandates a migration to Spring Framework 3 and Java 5 (1.5) or greater. Be aware that in many cases, migrating these other components may have a greater impact on your application than the upgrade of Spring Security!

Enhancements in Spring Security 3

Significant enhancements in Spring Security 3 over Spring Security 2 include the following:

- The addition of **Spring Expression Language** (**SpEL**) support for access declarations, both in URL patterns and method access specifications, which we have covered in *Chapter 2, Getting Started in Spring Security* and *Chapter 5, Fine-Grained Access Control*.

- Additional fine-grained configuration around authentication and accessing successes and failures, which we have covered briefly in Chapter 2, Chapter 5, and in detail in *Chapter 6, Advanced Configuration and Extension*.

- Enhanced capabilities of method access declaration, including annotation-based pre-and post-invocation access checks and filtering, as well as highly configurable `security` namespace XML declarations for custom backing bean behavior. These capabilities are examined in Chapter 5.

- Fine-grained management of session access and concurrency control using the `security` namespace, covered in Chapter 6.

- Noteworthy revisions to the ACL module, with the removal of the legacy ACL code in `o.s.s.acl` and some important issues with the ACL framework are addressed. ACL configuration and support is reviewed in *Chapter 7, Access Control Lists*.

- Support for OpenID Attribute Exchange, and other general improvements to the robustness of OpenID, illustrated in *Chapter 8, Opening up to OpenID*.

- New Kerberos and SAML single sign-on support through the Spring Security Extensions project that we discussed in *Chapter 12, Spring Security Extensions*.

Other more innocuous changes encompassed a general restructuring and cleaning up of the codebase and the configuration of the framework, such that the overall structure and usage makes much more sense. The authors of Spring Security have made efforts to add extensibility where none previously existed, especially in the areas of login and URL redirection.

If you are already working in a Spring Security 2 environment, you may not find compelling reasons to upgrade if you aren't pushing the boundaries of the framework. However, if you have found limitations in the available extension points, code structure, or configurability of Spring Security 2, you'll welcome many of the minor changes that we discuss in detail in the remainder of this chapter.

Changes to configuration in Spring Security 3

Many of the changes in Spring Security 3 will be visible in the `security` namespace style of configuration. Although this chapter cannot cover all of the minor changes in detail, we'll try to cover those changes that will be most likely to affect you as you move to Spring Security 3.

Rearranged AuthenticationManager configuration

The most obvious changes in Spring Security 3 deal with the configuration of the `AuthenticationManager` and any related `AuthenticationProvider` elements. In Spring Security 2, the `AuthenticationManager` and `AuthenticationProvider` configuration elements were completely disconnected—declaring an `AuthenticationProvider` didn't require any notion of an `AuthenticationManager` at all.

```
<authentication-provider>
  <jdbc-user-service data-source-ref="dataSource"
  />
</authentication-provider>
```

In Spring Security 2 to declare the `<authentication-manager>` element as a sibling of any `AuthenticationProvider`.

```
<authentication-manager alias="authManager"/>
<authentication-provider>
  <jdbc-user-service data-source-ref="dataSource"/>
</authentication-provider>
<ldap-authentication-provider server-ref="ldap://localhost:10389/"/>
```

In Spring Security 3, all `AuthenticationProvider` elements must be the children of `<authentication-manager>` element, so this would be rewritten as follows:

```
<authentication-manager alias="authManager">
  <authentication-provider>
    <jdbc-user-service data-source-ref="dataSource"
    />
  </authentication-provider>
  <ldap-authentication-provider server-ref=
    "ldap://localhost:10389/"/>
</authentication-manager>
```

Of course, this means that the `<authentication-manager>` element is now required in any security namespace configurations.

If you had defined a custom `AuthenticationProvider` in Spring Security 2, you would have decorated it with the `<custom-authentication-provider>` element as part of its bean definition, for example, the custom `AuthenticationProvider` that we implemented in Chapter 6:

```
<bean id="signedRequestAuthenticationProvider"
  class="com.packtpub.springsecurity.security
  .SignedUsernamePasswordAuthenticationProvider">
  <security:custom-authentication-provider/>
  <property name="userDetailsService" ref="userDetailsService"/>
<!-- ... -->
</bean>
```

While moving this custom `AuthenticationProvider` to Spring Security 3, we would remove the decorator element and instead configure the `AuthenticationProvider` as we saw in Chapter 3, using the `ref` attribute of the `<authentication-provider>` element as follows:

```
<authentication-manager alias="authenticationManager">
  <authentication-provider ref=
  "signedRequestAuthenticationProvider"/>
</authentication-manager>
```

Of course, the source code of our custom provider would change due to class relocations and renaming in Spring Security 3—look later in the chapter for basic guidelines, and in the code download for this chapter to see a detailed mapping.

New configuration syntax for session management options

In addition to continuing support for the session fixation and concurrency control features from prior versions of the framework, Spring Security 3 adds new configuration capabilities for customizing URLs and classes involved in session and concurrency control management, as described in detail in Chapter 6. If your older application was configuring session fixation protection or concurrent session control, the configuration settings have a new home in the `<session-management>` directive of the `<http>` element.

In Spring Security 2, these options would be configured as follows:

```
<http ... session-fixation-protection="none">
<!-- ... -->
  <concurrent-session-control exception-if-maximum-exceeded
```

```
    ="true" max-sessions="1"/>
  </http>
```

The analogous configuration in Spring Security 3 removes the `session-fixation-protection` attribute from the `<http>` element, and consolidates as follows:

```
<http ...>
  <session-management session-fixation-protection="none">
    <concurrency-control error-if-maximum-exceeded
      ="true" max-sessions="1"/>
  </session-management>
</http>
```

You can see that the new logical organization of these options is much more sensible and leaves room for future expansion.

Changes to custom filter configuration

Many users of Spring Security 2 have developed custom authentication filters (or other filters to alter the flow of a secured request). As with custom authentication providers, such filters were previously indicated through decoration of a bean with the `<custom-filter>` element. This made it a bit difficult in some cases to tie the configuration of a filter directly to the Spring Security configuration.

Let's see an example of the configuration for the signed request header filter from Chapter 6, as applied to a Spring Security 2 environment.

```
<bean id="requestHeaderFilter" class="com.packtpub
  .springsecurity.security.RequestHeaderProcessingFilter">
  <security:custom-filter after="AUTHENTICATION_PROCESSING_FILTER"/>
  <property name="authenticationManager"
    ref="authenticationManager"/>
</bean>
```

Contrast this with the same configuration from Spring Security 3, and you can see that the bean definition and security wiring are done independently. The custom filter is declared within the `<http>` element as follows:

```
<http ...>
<!-- ... -->
  <custom-filter ref="requestHeaderFilter"
    before="FORM_LOGIN_FILTER"/>
<!-- ... -->
</http>
```

The bean declaration remains the same as in Spring Security 2, although, as you'd expect, the code for a custom filter is quite different. We've included the sample code—using Spring Security 2 classes—for this filter along with this chapter, to provide you with some guidance as to what a custom filter should look like before and after translation to Spring Security 3.

Also, the logical filter names for some of the filters have changed in Spring Security 3. We present the list of changes in the following list and a full table for reference is provided in *Appendix*, *Additional Reference Material*:

Spring Security 2	Spring Security 3
SESSION_CONTEXT_INTEGRATION_FILTER	SECURITY_CONTEXT_FILTER
CAS_PROCESSING_FILTER	CAS_FILTER
AUTHENTICATION_PROCESSING_FILTER	FORM_LOGIN_FILTER
OPENID_PROCESSING_FILTER	OPENID_FILTER
BASIC_PROCESSING_FILTER	BASIC_AUTH_FILTER
NTLM_FILTER	Removed in Spring Security 3

You must make these changes in your configuration files in addition to relocating the `<custom-filter>` element.

Changes to CustomAfterInvocationProvider

One final bean decoration from Spring Security 2 has been replaced by a straight, inline element reference, a `CustomAfterInvocationProvider` declared by the `<custom-after-invocation-provider>` element.

```
<bean id="customAfterInvocationProvider"
  class="com.packtpub.springsecurity.security
  .CustomAfterInvocationProvider">
  <security:custom-after-invocation-provider/>
</bean>
```

Similar to what we saw with the other bean decorators from Spring Security 2, in Spring Security 3, this element has been moved within the `<global-method-security>` declaration, with a simple bean reference.

```
<global-method-security ...>
  <after-invocation-provider ref="customAfterInvocationProvider"/>
</global-method-security>
```

We've explored this and other aspects of method security in Chapter 5, including some of the interesting new options for method security added in Spring Security 3.

Minor configuration changes

The following points will briefly state the additional changes in configuration attributes between Spring Security 2 and 3:

- While using the `auto-config` attribute in Spring Security 3, remember me services are no longer configured by default. You will need to explicitly add the `<remember-me>` declaration within your `<http>` element.

- For LDAP configuration, the default value for `group-search-base-attribute` (used in LDAP authorities search) changed in Spring Security 3 from `ou=Groups` to an empty string (the root of the LDAP tree). We used this attribute in *Chapter 9*, *LDAP Directory Services*.

- The `<filter-invocation-definition-source>` decoration element, used for configuring a filter chain manually, has been renamed in Spring Security 3 to `<filter-security-metadata-source>`. We performed this type of configuration in the explicit bean configuration in Chapter 6.

- The attribute `exception-if-maximum-exceeded`, related to the `<concurrent-session-control>` element, has been moved and renamed in Spring Security 3 to `error-if-maximum-exceeded`, on the new `<concurrency-control>` element.

- While using the in memory DAO `UserDetailsService`, the `password` attribute is no longer required in Spring Security 3 when declaring users in the configuration file.

- Built-in support for NTLM authentication has been removed in Spring Securitiy 3 and is intended to be replaced with Kerberos authentication (please refer to Chapter 12 for details on configuring Kerberos). This remains a point of concern among upgrading users and some activity exists in the Spring Security community to retrofit Spring Security 3 with NTLM support. It is possible that, by the time the book has been published, a Spring Security Extensions project may exist to support NTLM in Spring Security 3.

The remainder of the changes in the XML configuration dialect for Spring Security 3 represent additions to functionality, and will not cause migration issues for existing applications configured with the `security` namespace.

Changes to packages and classes

Although in very straightforward applications of Spring Security 2, the locations of classes in packages may not matter, most applications of Spring Security don't end up being free from some kind of ties with the underlying code. So, we felt it would be helpful to point you in the direction of many of the overall package migrations and class renames that occurred between Spring Security 2 and 3.

Wherever possible, we have tried to map classes as accurately as we could—an overview of the major package moves are provided here, and (should you need it) a more comprehensive list is provided as a download with the source code. The table indicates the biggest relocations of classes from Spring Security 2 to Spring Security 3—we've truncated the table to include majority of changes you're likely to see.

Number of classes	Location in Spring 2	Location in Spring 3
13	o.s.s	o.s.s.authentication
13	o.s.s.acls	o.s.s.acls.model
13	o.s.s.event.authentication	o.s.s.authentication.event
12	o.s.s.vote	o.s.s.access.vote
11	o.s.s.ui.rememberme	o.s.s.web.authentication.rememberme
10	o.s.s.providers.jaas	o.s.s.authentication.jaas
10	o.s.s.securechannel	o.s.s.web.access.channel
10	o.s.s.userdetails.ldap	o.s.s.ldap.userdetails
9	o.s.s.providers.encoding	o.s.s.authentication.encoding
8	o.s.s.config	o.s.s.config.authentication
8	o.s.s.util	o.s.s.web.util
7	o.s.s.config	o.s.s.config.http
7	o.s.s.context	o.s.s.core.context
7	o.s.s.userdetails	o.s.s.core.userdetails
6	o.s.s	o.s.s.access
6	o.s.s.afterinvocation	o.s.s.acls.afterinvocation
6	o.s.s.event.authorization	o.s.s.access.event
6	o.s.s.util	o.s.s.web
5	o.s.s.annotation	o.s.s.access.annotation
5	o.s.s.authoritymapping	o.s.s.core.authority.mapping
5	o.s.s.providers	o.s.s.authentication
5	o.s.s.token	o.s.s.core.token
5	o.s.s.ui	o.s.s.web.authentication

If it appears to you that classes moved substantially, you are correct! Very few classes were untouched in the overall package reorganization as part of Spring Security 3. Hopefully, this overview points you in the right direction for classes that you may be looking for. Again, please consult the downloads for this chapter to review a detailed class-by-class mapping.

The benefit to this reorganization is that the entire framework is now much more modular, and fits into discrete JAR files containing only particular elements of functionality, as indicated in the following table (we used `nnn` in place of the release number):

JAR name	Functionality
`spring-security-acl-nnn.jar`	ACL support (see Chapter 7)
`spring-security-cas-client-nnn.jar`	CAS support (see Chapter 10)
`spring-security-config-nnn.jar`	Overall configuration support
`spring-security-core-nnn.jar`	Core framework and classes
`spring-security-ldap-nnn.jar`	LDAP support (see Chapter 9)
`spring-security-openid-nnn.jar`	OpenID support (see Chapter 8)
`spring-security-taglibs-nnn.jar`	JSP Tag Library support (see Chapters 3, 5, and 7)
`spring-security-web-nnn.jar`	Web tier support

This modularization means that, for example, it is possible to deploy Spring Security to a non-web application without any web dependencies (`spring-security-config` and `spring-security-core` might even be enough for your needs).

Summary

This chapter reviewed the major and minor changes that you will find when upgrading an existing Spring Security 2 project to Spring Security 3. In this chapter we have:

- Reviewed the significant enhancements to the framework that are likely to motivate an upgrade
- Examined upgrade requirements, dependencies, and common types of code and configuration changes that will prevent applications from working post-upgrade
- Investigated (at a high level) the overall code-reorganization changes that the Spring Security authors made as part of the codebase restructuring

If this is the first chapter you've read, we hope that you return to the rest of the book, using this chapter as a guide, to allow your upgrade to Spring Security 3 to proceed as smoothly as possible!

Additional Reference Material

In this appendix, we will cover some reference material which we feel is helpful (and largely undocumented), but too comprehensive to insert into the text of the chapters.

Getting started with JBCP Pets sample code

As we described in *Chapter 1, Anatomy of an Unsafe Application,* we have made the assumption that you have access to the Eclipse 3.4 (or 3.5) IDE, with the Web Tools Package (WTP). The sample code is structured into different ZIP files, one per chapter, with a single large dependencies ZIP file containing all the dependencies required to compile and run the sample application (note that these may be out of date with the most current versions of Spring Security by the time you read this book, but as the combination of sample code and dependencies are a static snapshot, they are guaranteed to continue to work as they are in perpetuity).

We would suggest that you create a new Eclipse workspace for each chapter, this way you can switch among workspaces without having to worry about opening and closing projects.

The following steps should help you set up the new workspace. First, we will import the Dependencies project into the workspace:

- Select the **File** menu, and then the **Import...** option. Choose the **General** folder, and **Existing Projects into Workspace** and click on **Next**.

- Ensure the **Select root directory** option is checked, and click the **Browse...** button next to this text box. Locate the directory where you expanded the `Dependencies.zip` file and click on **OK**.

- You should see the `Dependencies` project listed. Click on **Finish**.

Next, we'll have to import an individual chapter source code ZIP file. Let's assume you've unzipped the source code for *Chapter 2, Getting Started with Spring Security,* into a directory.

- Select the **File** menu, and the **Import...** option. Choose the **General** folder, and **Existing Projects into Workspace**. Click on **Next**.
- Ensure the **Select root directory** option is checked, and then click the **Browse...** button next to this text box. Locate the directory where you expanded the Chapter 2 source code ZIP file and click on **OK**.
- You should see the JBCPPets project and the Servers project listed. Ensure both are checked and click **Finish**.

Finally, we will need to deploy the JBCP Pets web application to an instance of Tomcat (or your favorite application server).

- Right-click the JBCPPets project and then select the **Run As** menu. Select the **Run on Server** item from the submenu.
- You may be required at this point to create a new application server instance. Simply follow the prompts for your application server until the web application is deployed.

At this point, you should be running the JBCP Pets application! If you run into trouble, review the following checklist:

- Does Eclipse list any build errors? If there are any Java errors or classpath errors, these are true errors and should be corrected. In some cases, the Spring IDE plugin will report errors or warnings that are spurious.
- Review the application server startup console in Eclipse for errors indicating an unsuccessfully deployed web application. The most common issue is a missing classpath item, or forgetting to add all classpath items to the **Java EE Module Dependencies** tab of the JBCPPets project. Although this should be done for you, we can't be sure that this will work in every version of Eclipse!

Please contact us if you have any trouble with the sample code— it's vitally important for you to understand the code and concepts presented!

Available application events

The following table, referenced in *Chapter 6, Advanced Configuration and Extension,* lists the full set of events broadcasted by various Spring Security elements and also provides list of authentication exceptions referenced in Chapter 2. We've omitted the package names for brevity, because all events saved are in the o.s.s.authentication.event (for authentication-related events) and o.s.s.access.event (for authorization-related events).

Class Name	When is it fired?	Maps to Exception
`AbstractAuthenticationEvent`	Common superclass of all authentication events. Note that this is an abstract exception, which is never thrown (although it may be caught).	
`AbstractAuthenticationFailureEvent`	Common superclass of all authentication failure events. Note that this is an abstract exception, which is never thrown (although it may be caught).	

`AuthenticationFailureBadCredentials Event`	When credentials presented (such as username and password) are not valid. It can be used to (intentionally) obscure `UsernameNotFound Exception`.	`BadCredentials Exception`
		
		`UsernameNotFound Exception`

`AuthenticationFailureConcurrent LoginEvent`	When concurrent session maximum is exceeded.	`ConcurrentLogin Exception`
`AuthenticationFailureCredentials ExpiredEvent`	When `UserDetails` indicates that user credentials are expired.	`CredentialsExpired Exception`
`AuthenticationFailureDisabledEvent`	When `UserDetails` indicates that user credentials are disabled.	`DisabledException`
`AuthenticationFailureExpiredEvent`	When `UserDetails` indicates that user account is expired.	`AccountExpired Exception`
`AuthenticationFailureLockedEvent`	When `UserDetails` indicates that user account is locked.	`LockedException`
`AuthenticationFailureProvider NotFoundEvent`	Configuration error, when an `Authentication Provider` can't be found to authenticate a user request.	`ProviderNotFound Exception`
`AuthenticationFailureProxy UntrustedEvent`	Thrown when a CAS proxy ticket is not trusted.	

Class Name	When is it fired?	Maps to Exception
`AuthenticationFailureService` `ExceptionEvent`	Generic exception thrown when an underlying service (DAO Provider, and so on) fails.	`Authentication` `ServiceException`
`AuthenticationSuccessEvent`	When a user is successfully authenticated.	
`AuthenticationSwitchUserEvent`	When a user successfully performs a switch user action.	
`InteractiveAuthenticationSuccess` `Event`	When a user is successfully authenticated by presenting a full set of credentials (similar to `IS_FULLY_` `AUTHENTICATED` `GrantedAuthority` criteria).	
`AbstractAuthorizationEvent`	Common superclass of all authorization events.	
`AuthenticationCredentialsNot` `FoundEvent`	When a user has not been authenticated, and tries to invoke a method requiring access check.	
`AuthorizationFailureEvent`	When an access check before or after a method fails.	
`AuthorizedEvent`	When an access check before or after a method succeeds.	
`PublicInvocationEvent`	When a non-authenticated request on a secured object succeeds.	
`SessionCreationEvent`	When `HttpSession` is created.	
`SessionDestroyedEvent`	When `HttpSession` is destroyed.	

Spring Security virtual URLs

The following URLs are treated as virtual URLs by Spring Security, and watched (and processed) as part of servlet filter processing, independently of your code. Remember that these URLs are relative to your web application's context root.

- `/j_spring_security_check`—checked by `UsernamePasswordAuthenticationFilter` for username/password form authentication.
- `/j_spring_openid_security_check`—checked by `OpenIDAuthenticationFilter` for OpenID returning authentication (from the OpenID provider).
- `/j_spring_cas_security_check`—used by CAS authentication upon return from CAS SSO login.
- `/spring_security_login`—the URL used by the `DefaultLoginPageGeneratingFilter` when configured to auto-generate a login page.
- `/j_spring_security_logout`—used by `LogoutFilter` to detect a log out action.
- `/saml/SSO`—used by the Spring Security SAML SSO extension `SAMLProcessingFilter` to process a SAML SSO sign-on request.
- `/saml/logout`—used by the Spring Security SAML SSO extension `SAMLLogoutFilter` to process a SAML SSO sign-out request.
- `/j_spring_security_switch_user`—used by the `SwitchUserFilter` to switch users to another user.
- `/j_spring_security_exit_user`—used to exit the switch user functionality.

Note that some of this functionality was not covered in this book, but we've included everything for the sake of completeness.

Method security explicit bean configuration

The `dogstore-explicit-base.xml` file in the Chapter 6 source code contains the full set of bean declarations listed here. We chose not to include it in Chapter 6 itself, because it doesn't have much associated explanation required (refer to *Chapter 5, Fine-Grained Access Control*, for the roles of the individual beans involved).

Here is the full configuration listing required to enable method security through Spring Bean configuration:

```xml
<!-- ************************************************ -->
<!-- Method Authorization                            -->
<!-- ************************************************ -->

<bean class="org.springframework.security.access.intercept.
aopalliance.MethodSecurityInterceptor" id="methodSecurityInterceptor">
  <property name="accessDecisionManager" ref="methodAccessDecisionMan
ager"/>
  <property name="authenticationManager" ref="customAuthenticationMan
ager"/>
  <property name="securityMetadataSource" ref="delegatingMetadataSour
ce"/>
  <property name="afterInvocationManager" ref="afterInvocationManager
"/>
</bean>

<bean class="org.springframework.security.access.intercept.
aopalliance.MethodSecurityMetadataSourceAdvisor" id="methodSecurityMet
adataSourceAdvisor">
  <constructor-arg value="methodSecurityInterceptor"/>
  <constructor-arg ref="delegatingMetadataSource"/>
</bean>
<bean class="org.springframework.aop.framework.autoproxy.
DefaultAdvisorAutoProxyCreator" id="defaultAdvisorAutoProxyCreator">
  <property name="beanName" value="methodSecurityMetadataSourceAdviso
r"/>
</bean>


<bean class="org.springframework.security.access.intercept.
AfterInvocationProviderManager" id="afterInvocationManager">
  <property name="providers">
    <list>
      <ref local="postAdviceProvider"/>
    </list>
  </property>
</bean>

<bean class="org.springframework.security.access.vote.
AffirmativeBased" id="methodAccessDecisionManager">
  <property name="decisionVoters">
    <list>
```

```
        <ref bean="preAdviceVoter"/>
        <ref bean="roleVoter"/>
        <ref bean="authenticatedVoter"/>
        <ref bean="jsr250Voter"/> <!-- For JSR 250 Method Annotations
-->
    </list>
  </property>
</bean>


<!-- Overall Delegating Metadata Source -->
<bean class="org.springframework.security.access.method.
DelegatingMethodSecurityMetadataSource" id="delegatingMetadataSource">
  <property name="methodSecurityMetadataSources">
    <list>
     <ref local="prePostMetadataSource"/>
      <ref local="securedMetadataSource"/>
      <ref local="jsr250MetadataSource"/>
    </list>
  </property>
</bean>


<!-- JSR 250 Method Voters -->
<bean class="org.springframework.security.access.annotation.
Jsr250MethodSecurityMetadataSource" id="jsr250MetadataSource"/>
<bean class="org.springframework.security.access.annotation.
Jsr250Voter" id="jsr250Voter"/>


<!-- Spring @Secured Beans -->
<bean class="org.springframework.security.access.annotation.
SecuredAnnotationSecurityMetadataSource" id="securedMetadataSource"/>


<!-- @Pre/@Post Method Advice Voters -->
<bean class="org.springframework.security.access.prepost.
PreInvocationAuthorizationAdviceVoter" id="preAdviceVoter">
  <constructor-arg ref="exprPreInvocationAdvice"/>
</bean>
<bean class="org.springframework.security.access.prepost.
PostInvocationAdviceProvider" id="postAdviceProvider">
  <constructor-arg ref="exprPostInvocationAdvice"/>
</bean>
<bean class="org.springframework.security.access.prepost.
PrePostAnnotationSecurityMetadataSource" id="prePostMetadataSource">
  <constructor-arg ref="exprAnnotationAttrFactory"/>
</bean>
```

```
<!-- @Pre/@Post Method Expression Handler -->
<bean class="org.springframework.security.access.expression.method.
DefaultMethodSecurityExpressionHandler" id="methodExprHandler"/>
<bean class="org.springframework.security.access.expression.method.
ExpressionBasedPreInvocationAdvice" id="exprPreInvocationAdvice">
  <property name="expressionHandler" ref="methodExprHandler"/>
</bean>
<bean class="org.springframework.security.access.expression.method.
ExpressionBasedPostInvocationAdvice" id="exprPostInvocationAdvice">
  <constructor-arg ref="methodExprHandler"/>
</bean>
<bean class="org.springframework.security.access.expression.method.
ExpressionBasedAnnotationAttributeFactory" id="exprAnnotationAttrFact
ory">
  <constructor-arg ref="methodExprHandler"/>
</bean>
```

Please note that explicit bean configuration is very tightly coupled to the version of Spring Security that you are using (as we also noted in Chapter 6). Refer to the `o.s.s.config.method.GlobalMethodSecurityBeanDefinitionParser` if you are encountering issues with the beans listed here in your version of Spring Security.

This configuration enables JSR-250 annotations, `@Secured`, and `@Pre/@Post` annotations. You may comment or remove the relevant supporting beans (for example `@Secured`) if you are not using them. Remember that both the `SecurityMetadataSource` and `AccessDecisionVoter` should be removed.

Logical filter names migration reference

As discussed in *Chapter 13, Migration to Spring Security 3*, many of the logical filter names (used in the `<custom-filter>` element) changed when we migrate from Spring Security 2 and 3. We present a full table of the changes here, to ease your migration of custom filter configuration from Spring Security 2 to 3:

Spring Security 2	Spring Security 3
CHANNEL_FILTER	CHANNEL_FILTER
CONCURRENT_SESSION_FILTER	CONCURRENT_SESSION_FILTER
SESSION_CONTEXT_INTEGRATION_ FILTER	SECURITY_CONTEXT_FILTER
LOGOUT_FILTER	LOGOUT_FILTER
PRE_AUTH_FILTER	PRE_AUTH_FILTER
CAS_PROCESSING_FILTER	CAS_FILTER
AUTHENTICATION_PROCESSING_FILTER	FORM_LOGIN_FILTER
OPENID_PROCESSING_FILTER	OPENID_FILTER
LOGIN_PAGE_FILTER is not present in Spring Security 2	LOGIN_PAGE_FILTER
DIGEST_AUTH_FILTER is not present in Spring Security 2	DIGEST_AUTH_FILTER
BASIC_PROCESSING_FILTER	BASIC_AUTH_FILTER
REQUEST_CACHE_FILTER is not present in Spring Security 2	REQUEST_CACHE_FILTER
SERVLET_API_SUPPORT_FILTER	SERVLET_API_SUPPORT_FILTER
REMEMBER_ME_FILTER	REMEMBER_ME_FILTER
ANONYMOUS_FILTER	ANONYMOUS_FILTER
SESSION_MANAGEMENT_FILTER is not present in Spring Security 2	SESSION_MANAGEMENT_FILTER
EXCEPTION_TRANSLATION_FILTER	EXCEPTION_TRANSLATION_FILTER
NTLM_FILTER	NTLM_FILTER is removed in Spring Security 3
FILTER_SECURITY_INTERCEPTOR	FILTER_SECURITY_INTERCEPTOR
SWITCH_USER_FILTER	SWITCH_USER_FILTER

Index

C

PACKT open source *
community experience distilled

Thank you for buying
Spring Security 3

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

Spring Persistence with Hibernate

ISBN: 978-1-849510-56-1   Paperback: 460 pages

Build robust and reliable persistence solutions for your enterprise Java application

1. Get to grips with Hibernate and its configuration manager, mappings, types, session APIs, queries, and much more

2. Integrate Hibernate and Spring as part of your enterprise Java stack development

3. Work with Spring IoC (Inversion of Control), Spring AOP, transaction management, web development, and unit testing considerations and features

Spring Python 1.1

ISBN: 978-1-849510-66-0   Paperback: 380 pages

Build powerful web applications, quickly and cleanly, with the Django application framework

1. Maximize the use of Spring features in Python and develop impressive Spring Python applications

2. Explore the versatility of Spring Python by integrating it with frameworks, libraries, and tools

3. Discover the non-intrusive Spring way of wiring together Python components

4. Packed with hands-on-examples, case studies, and clear explanations for better understanding
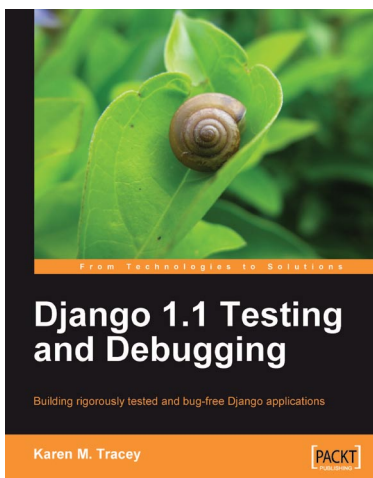
Please check **www.PacktPub.com** for information on our titles