THE EXPERT'S VOICE® IN SPRING

Pro Spring Security

SECURING YOUR ENTERPRISE AND WEB SPRING AND GRAILS APPLICATIONS

Carlo Scarioni

Apress®

Download from Wow! eBook <www.wowebook.com>

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



Apress®

Contents at a Glance

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

■ Chapter 1: The Scope of Security 1

■ Chapter 2: Introducing Spring Security 9

■ Chapter 3: Spring Security Architecture and Design 27

■ Chapter 4: Web Security 57

■ Chapter 5: Securing the Service Layer 111

■ Chapter 6: Configuring Alternative Authentication Providers 153

■ Chapter 7: Business Object Security with ACLs 205

■ Chapter 8: Customizing and Extending Spring Security 237

■ Chapter 9: Integrating Spring Security with Other Frameworks and Languages 273

Index 311

Introduction

Denying the impact of the Spring Framework in the Java world would be simply impossible. Spring has brought so many advantages to the Java developer that I could say it has made better developers of all of us. The good ones, the average ones. All of us.

Spring's core building blocks of Dependency Injection and Aspect Oriented Programming are widely applicable to many business and infrastructure concerns, and certainly application security can benefit from these core functionalities. So this is Spring Security: an application-level security framework built on top of the powerful Spring Framework that deals mainly with the core security concepts of *authentication* and *authorization*.

Spring Security aims to be a full-featured security solution for your Java applications. Although its main focus is on Web applications and the Java programming language, you will see that it goes beyond these two domains.

What I wanted to do in writing this book was to expose some of the internal works of Spring Security along with the standard explanations of how to use certain features. My idea is to teach beyond the basics of how to do something in particular, and instead focus on the plumbing inside the framework. For me, this is the best way of learning something: actually seeing how it is built in the core. That's not to say, of course, that the book doesn't cover basic setups and give quick, practical advice on using the framework, because it certainly does. The point I'm making is that instead of saying, "Use this to do that," I normally say, "This works like this… and this allows you to…." This is a point of view that only tools like Spring afford (because they are open source).

With that said, I suggest that the best way to use this book is to have the Spring Security source code checked out on your computer and go through the examples with both the code from the book and the code from Spring Security itself. This will not only help you understand each concept as it is introduced, but will also teach more than one good programming trick and good practice. I recommend this approach to studying any software whenever you have the chance. If the source code is out there, grab it. Sometimes a couple lines of code teach more than a thousand words.

Who This Book Is For

This book is written mainly for Java developers who use Spring in their work and need to add security to their applications in a way that leverages Spring's proven concepts and techniques. The book will also be helpful to developers who want to add Web-layer security to their applications, even if those applications are not fully Spring powered at their core. The book assumes you have knowledge of Java and some of its tools and libraries, such as Servlets and Maven. It also assumes that you know what you want to use security for and in what context you want to use it. This means, for example, I won't explain protocols like LDAP in much depth; instead, I'll concentrate on showing you how to integrate Spring Security with an LDAP user store. An in-depth knowledge of Spring is not essential because many of the concepts are introduced as we go along, but the more you understand about Spring, the more you are likely to get out of this book.

How This Book Is Structured

The book is divided into nine chapters that embody a progressive study of Spring Security. Starting from a summary of basic applications and an explanation of how the framework is structured, the content moves on to more advanced topics, such as using Spring Security in different JVM languages. The book follows a sequence that corresponds to the way this framework is normally used in real life.

The chapters in the book include the following:

- **Chapter 1:** Introduces security in general and how to approach security problems at the application level.
- **Chapter 2:** Introduces Spring Security with a simple example application that secures Web access at the URL level.
- **Chapter 3:** Provides a full introduction to the architecture of Spring Security. The chapter covers its main components and how they interact with each other.
- **Chapter 4:** Gives in-depth coverage of the web-layer security options available in Spring Security.
- **Chapter 5:** Presents, as a counterpart to Chapter 4, full coverage of service-layer security.
- **Chapter 6:** Covers a wide array of authentication providers, including LDAP and JASS, that can be plugged into Spring Security.
- **Chapter 7:** Covers access control lists (ACL) that are used to secure individual domain objects and how they fit into the general security concerns.
- **Chapter 8:** Explains how to extend the core Spring Security functionality by making use of the many extension points supported by its modular architecture.
- **Chapter 9:** Shows how to integrate Spring Security with different Java frameworks and some important JVM programming languages.

Prerequisites

The examples in this book are all built with Java 7 and Maven 3. The latest Spring versions are used if possible. Spring Security 3.1.3 was the version used throughout the book. Jetty Web Server was used for the different web applications in the book, mainly through its Maven plugin. I worked mainly on my MacBook Air 2011 with 4 GBs of RAM. All the projects were developed using the IDE SpringSource Tool Suite.

You are free to use your own tools and operating system. Because everything is Java based, you should be able to compile your programs on any platform without problems.

Downloading the code

The code for the examples shown in this book is available on the Apress web site, `www.apress.com`. A link can be found on the book's information page under the Source Code/Downloads tab. This tab is located underneath the Related Titles section of the page.

Contacting the Author

You are more than welcome to send me any feedback regarding this book or any other subject I might help you with. You can contact me via my blog at `http://cscarioni.blogspot.com`, or you can send me an email at `carlo.scarioni@gmail.com`.

CHAPTER 1

The Scope of Security

Security. An incredibly overloaded word in the IT world. It means so many different things in so many different contexts, but in the end, it is all about protecting sensitive and valuable resources against malicious usage.

In IT, we have many layers of infrastructure and code that can be subject to malicious attacks, and arguably we should ensure that all these layers get the appropriate levels of protection.

Of course, the growth of the Internet and the pursuit of reaching more people with our applications have opened more and more doors to cyber criminals trying to access these applications in illegitimate ways.

It is also true that proper care is not always taken to ensure that a properly secured set of services is being offered to the public. And sometimes, even when good care is taken, some hackers are still smart enough to overcome security barriers that, superficially, appear adequate.

The three major security layers in an IT infrastructure are the network, the operating system, and the application itself.

The Network Security Layer

This layer is probably the most familiar one in the IT world. When people talk about IT security, they normally think of network-level security—in particular, security that uses firewalls.

Even though people often associate security with the network level, this is only a very limited layer of protection against attackers. Generally speaking, it can do no more than defend IP addresses and filter network packets addressed to certain ports in certain machines in the network.

This is clearly not enough in the vast majority of cases, as traffic at this level is normally allowed to enter the publicly open ports of your various exposed services with no restriction at all. Different attacks can be targeted at these open services, as attackers can execute arbitrary commands that could compromise your security constraints. There exist tools like the popular `nmap` (`http://nmap.org/`) that can be used to scan a machine to find open ports. The use of tools like this is an easy first step to take in preparing an attack, because well-known attacks can be used against such open ports if they are not properly secured.

A very important part of the network-layer security, in the case of web applications, is the use of Secure Sockets Layer (SSL) to encode all sensitive information sent along the wire, but this is related more to the network protocol at the application level than to the network physical level at which firewalls operate.

The Operating System Layer

This layer is probably the most important in the whole security schema, as a properly secured operating system (OS) environment could at least prevent a whole host machine from going down if a particular application is compromised.

If an attacker is somehow allowed to have unsecured access to the operating system, he can basically do whatever he wants—from spreading viruses to stealing passwords or deleting your whole server's data and making it unusable. Even worse perhaps, he could take control of your computer without you even noticing, and use it

to perform other malicious acts as part of a botnet. We can include in this layer the deployment model of the applications, as you need to know your operating system's permission scheme to ensure that you don't give your applications unnecessary privileges over your machine. Applications should run as isolated as possible from the other components of the host machine.

The Application Layer

The main focus of this book will be on this layer. The application security layer refers to all the constraints we establish in our applications to make sure that only the right people can do only the right things when working through the application.

Applications, by default, are open to countless avenues of attack. An improperly secured application can allow an attacker to steal information from the application, impersonate other users, execute restricted operations, corrupt data, gain access to operating system level, and perform many other malicious acts.

In this book, we will cover application-level security, which is the domain of Spring Security. Application-level security is achieved by implementing several techniques, and there are a few concepts that will help you understand better what the rest of the book will cover. These are the main concerns that Spring Security addresses to provide your applications with comprehensive protection against threats. In the following three subsections, I shall introduce

- Authentication
- Authorization
- ACLs

Authentication

The process of authentication allows an application to validate that a particular user is who she claims she is. In the authentication process, a user presents the application with information about herself (normally, a username and a password) that no one else knows. The application takes this information and tries to match it against information it has stored—normally, in a database or LDAP[1] (Lightweight Directory Access Protocol) server. If the information input by the user matches a record in the authentication server, the user is said to have successfully authenticated herself in the system. The application will normally create an internal abstraction representing this authenticated user in the system. Figure 1-1 shows the authentication mechanism.

[1]LDAP will be explained in some detail in Chapter 7, where various authentication providers are covered.

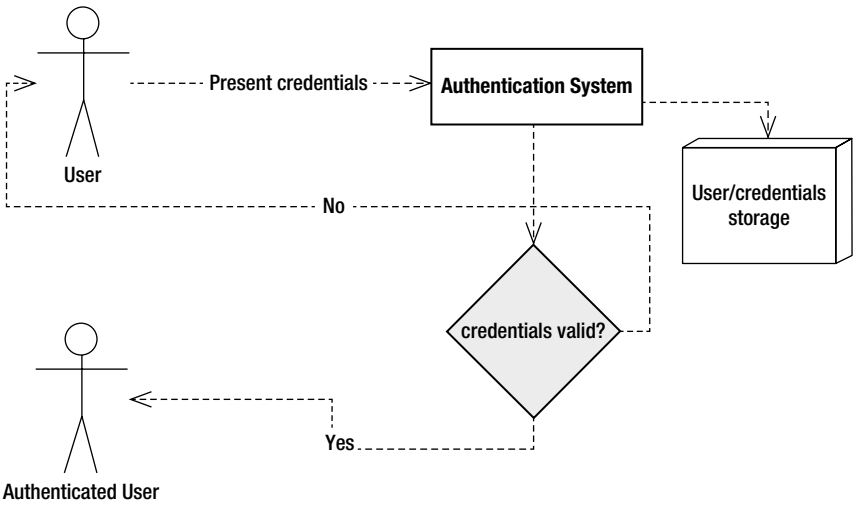


Figure 1-1. *Simple standard authentication mechanism*

Authorization

When a user is authenticated, that only means that the user is known to the system and has been recognized by it. It doesn't mean that the user is free to do whatever she wants in said system. The next logical step in securing an application is to determine which actions that user is allowed to perform, and which resources she has access to, and make sure that if the user doesn't have the proper permissions she cannot carry out that particular action. This is the work of the authorization process. In the most common case, the authorization process compares the user's set of permissions against the permissions required to execute a particular action in the application, and if a match is found, access is granted. On the other hand, if no match is found, access is denied. Figure 1-2 shows the authorization mechanism.

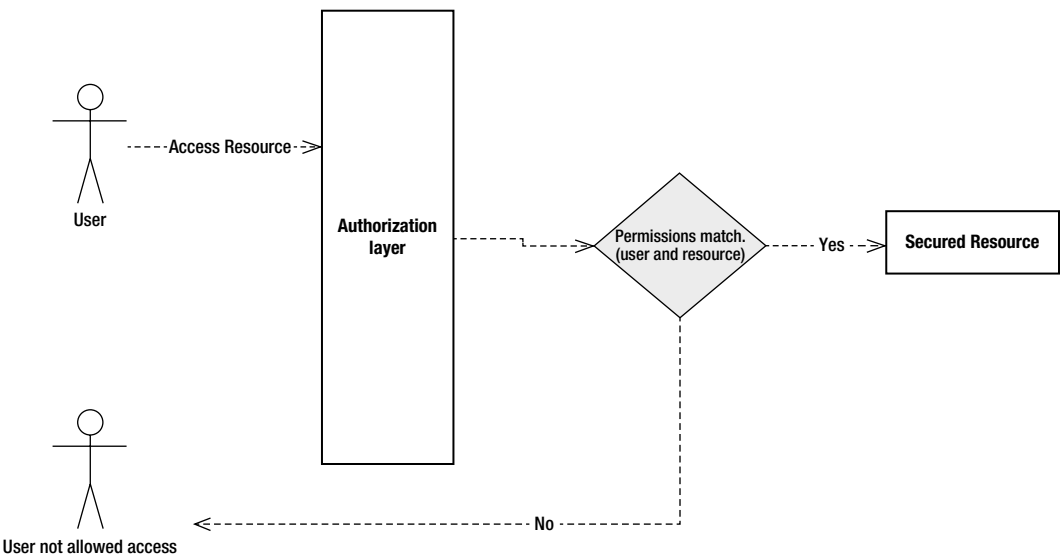


Figure 1-2. *Simple authorization process. The authenticated user tries to access a secured resource*

ACLs

Access control lists (ACLs) are part of the authorization process explained in the previous section. The key difference is that ACLs normally work at a finer grained level in the application. ACLs are simply a collection of mappings between resources, users, and permissions. With ACLs, you can establish rules like "User John has administrative permission on the blog post X" or "User Luis has read permission on blog post X." You can see the three elements: user, permission, and resource. Figure 1-2 shows how ACLs work, as they are just a special case of the general authorization process.

Authentication and Authorization: General Concepts

In this section, I shall introduce and explain some fundamental security concepts that you will be coming across frequently in the rest of the book:

- **User** The first step in securing a system from malicious attackers is to identify legitimate users and allow access to them alone. User abstractions are created in the system and given their own identity. These are the users that will later be allowed to use the system.
- **Credentials** Credentials are the way that a user proves who he is. Normally, in the shape of passwords (certificates are also a common way of presenting credentials), they are data that only the owner of it knows.
- **Role** In an application security context, a role can be seen as a logical grouping of users. This logical grouping is normally done so the grouped users share a set of permissions in the application to access certain resources. For example, all users with the role "admin" will have the same access and permissions to the same resources. Roles serve simply as a way to group permissions to execute determined actions, making users with those Roles inherit such permissions.
- **Resource** By a *resource*, I mean, in this context, any part of the application that we want to access and that needs to be properly secured against unauthorized access—for example, a URL, a business method, or a particular business object.
- **Permissions** Permissions refer to the access level needed to access a particular resource. For example, two users may be allowed to read a particular document, but only one of them is allowed to write to it. Permissions can apply either to individual users or to users that share a particular role.
- **Encryption** This allows you to encrypt sensible information (normally passwords, but it can be something else, like cookies) so as to make it incomprehensible to attackers even if they get access to the encrypted version. The idea is that you never store the plain text version of a password, but instead store an encrypted version so that nobody but the owner of such a password knows the original one. There are three main kinds of encryption algorithms:
    - **One-way encryption** These algorithms, referred as *hashing algorithms*, take an input string and generate an output number known as the *message digest*. This output number cannot be converted back into the original string. This is why the technique is referred to as *one-way encryption*. Here is the way to use it: A requesting client encrypts a string and sends the encrypted string to the server. The server may have access to the original information from a previous registration process, for example, and if it does, it could apply the same hash function to it. Then it compares the output from this hashing to the value sent by the client. If they match, the server validates the information. Figure 1-3 shows this scheme. Usually, the server doesn't even need the original data. It could simply store the hashed version and then compare it with the incoming hash from the client.

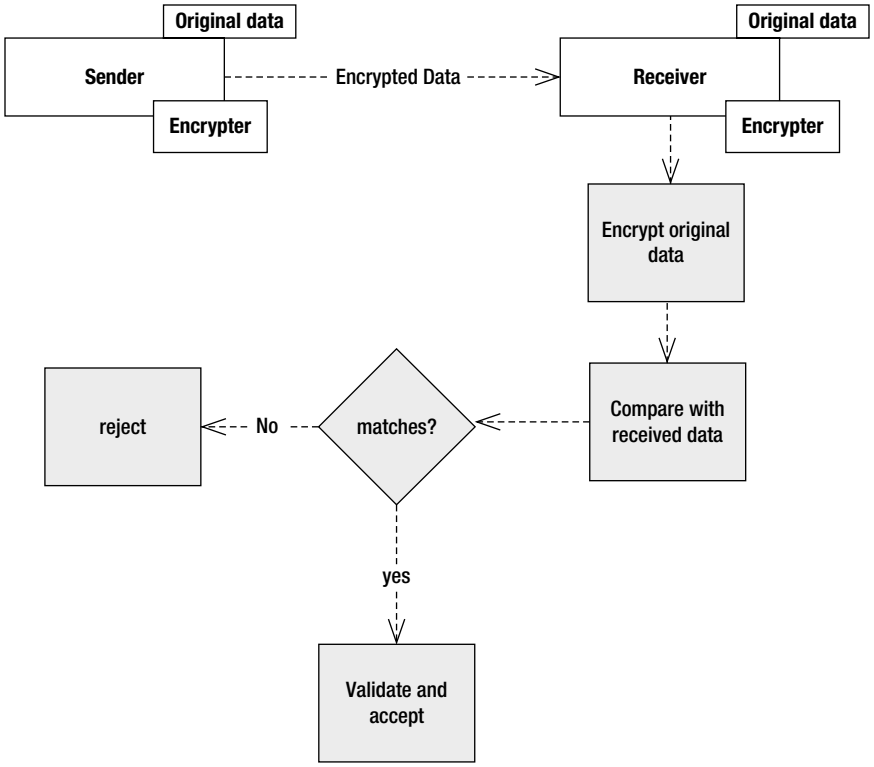


Figure 1-3. *One-way encryption or hashing*

- **Symmetric encryption** These algorithms provide two functions: encrypt and decrypt. A string of text is converted into an encrypted form and then can be converted back to the original string. In this scheme, a sender and a receiver share the same keys so that they can encrypt and decrypt messages on both ends of the communication. One problem with this scheme is how to share the key between the endpoints of the communication. A common approach is to use a parallel secure channel to send the keys. Figure 1-4 shows symmetric encryption at work.

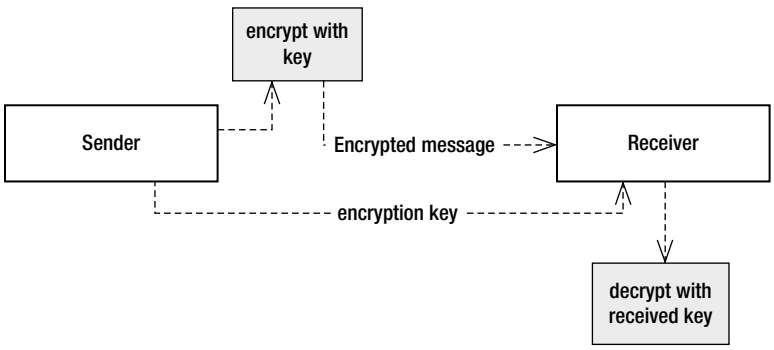


Figure 1-4. *Symmetric encryption. The two endpoints share the same encryption/decryption key*

- **Public key cryptography** These techniques are based on asymmetric cryptography. In this scheme, a different key is used for encryption than for decryption. These two keys are referred as the *public key*, which is used to encrypt messages, and the *private key*, which is used to decrypt messages. The advantage of this approach over symmetric encryption is that there is no need to share the decryption key, so no one but the intended receiver of the information is able to decrypt the message. So the normal scenario is the following:
- The intended recipient of messages shares its public key with everyone interested in sending information to it.
- The senders encrypt the information with the receiver's public key, and send the message.
- The receiver uses its private key to decrypt the message.
- No one else is able to decrypt the message as they don't have the receiver's private key.

Figure 1-5 shows the public key cryptography scheme.

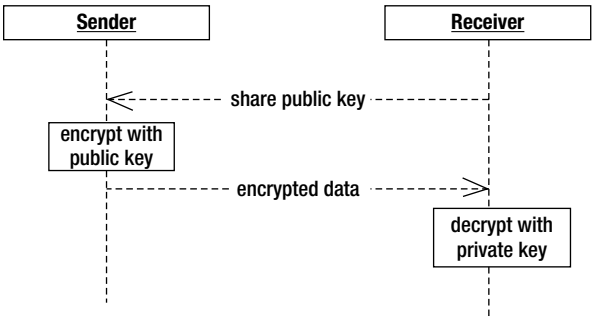


Figure 1-5. *Public key cryptography*

The use of encryption achieves, among other things, two other security goals:

- **Confidentiality** Potentially sensitive information belonging to one user, or group of users, should be accessible only to this user or group. Encryption algorithms are the main helper in achieving this goal.
- **Integrity** Data sent by a valid user shouldn't be altered by a third entity on its way to the server, or in its storage. This is normally accomplished through the use of one-way cryptographic algorithms that make it almost impossible to alter an input and produce a corrupted message whose encrypted hash is the same as the original message (thus deceiving the receiver into thinking it is valid).

What to Secure

Not every part of the application requires a strong security model, or even any security at all. If, for example, one part of your application is supposed to serve static content to everyone interested in it, you can simply serve this content. There probably are no security concerns to handle here.

Anyway, when starting to work on a new application, you should think about the security constraints that your application will have. You should think about concerns like those in the following list and whether or not they apply to your particular use case:

- **Identity management** More than likely, your application will need to establish the identities of the different users that will be using it. Usually, your application will do different things for different users, so you need a way to associate users with certain functionality. You also need to be sure to protect each user's identity information so that it can't be compromised.
- **Secured connections** In an Internet environment, where anyone in the world can potentially access your system and eavesdrop on other users' accessing your system, you most likely will want to secure the communication of sensitive data using some kind of transport layer security—for example, SSL.
- **Sensitive data protection** Sensitive data will need to be protected against malicious attacks. This applies both to the communication layer and to individual message transmission, as well as to credentials datastores. Encryption will be used in different layers to achieve the most secure application possible.

More Security Concerns

There are many more security concerns than the ones explained so far. Because this is a Spring Security book and not a general application-security book, we will cover only things related to Spring Security. However, I think it is important that you understand there are many more security concerns than those addressed directly by Spring Security. Following is a quick overview of some of the most common ones. This is only intended to make you aware of their existence, and I recommend you consult a different source (such as a general software security textbook) to gain a better understanding of all these concerns:

- **SQL (and other code) injection** Validating user input is a very important part of application security. If data is not validated, an attacker could potentially write any kind of string as input (including SQL or server-side code) and send that information to the server. If the server code is not properly written, the attacker could wreak significant havoc, as she could execute any arbitrary code on the server.
- **Denial of service attacks** These attacks consist of making the target system unresponsive to its intended users. This is normally done by saturating the server with requests so that it utilizes all the server's resources and makes it unresponsive to legitimate requests.
- **Cross-site scripting and output sanitation** A kind of injection can be done where the target is the client part of the application. The idea is that the attacker can make an application return malicious code inside the web pages returned, and thus execute it in the user's browser. This way, the attacker invisibly executes actions using the real user's authenticated session.

Download from Wow! eBook <www.wowebook.com>

Java Options for Security

Java and Java EE out-of-the-box security solutions are very comprehensive. They cover areas ranging from a low-level permission system, through cryptography APIs, to an authentication and authorization scheme.

The list of security APIs offered in Java is very extensive, as the following list of the main ones shows:

- **Java Cryptography Architecture (JCA)** This API offers support for cryptographic algorithms, including hash-digest and digital-signature support.
- **Java Cryptographic Extensions (JCE)** This API mainly provides facilities for the encryption and decryption of strings and also secret key generation for symmetric algorithms.
- **Java Certification Path API (CertPath)** This API provides comprehensive functionality for integrating the validation and verification of digital certificates into an application.
- **Java Secure Socket Extension (JSSE)** This API Provides a standardized set of features to offer support for SSL and TLS protocols, both client and server, in Java.
- **Java Authentication and Authorization Service (JAAS)** This API provides service for authentication and authorization in Java applications. It provides a pluggable system where authentication mechanisms can be plugged in independently to applications.

Spring Security's main concerns are in the authentication/authorization realm. So it overlaps mainly with the JAAS Java API, although they can be used together, as you will see later in the book. Most of the other APIs are leveraged in Spring Security. For example, CertPath is used in X509AuthenticationFilter and JCE is used in the spring-security-crypto module.

Summary

In this chapter, I introduced security from a general point of view. I explained in a very abstract way the main concerns in IT security and especially from an application point of view. I also described, very briefly, the main Java APIs that support security at different levels.

You can see that this chapter was a very quick overview of security concerns. It is beyond the scope of this book to go any further than this on general topics, although we will study some of them in more depth when they apply to Spring Security. Obviously this is nothing like a comprehensive software security guide, and if you are interested in learning more about software security in general you should consult the specialized literature. The next chapter will introduce Spring Security as such.

CHAPTER 2

Introducing Spring Security

In this chapter, you will learn what Spring Security is and how you can use it to address security concerns about your application. We'll build a simple application secured with Spring Security. We'll start with a Servlet-based web application without any security, and then we'll add security to it in a declarative, nonintrusive way.

Also in this chapter, we'll take a look at the framework's source code, how to build it, and the different modules that together form the powerful Spring Security project.

What Is Spring Security?

Spring Security is a framework dedicated to providing a full array of security services to Java applications in a developer-friendly and flexible way. It adheres to the well-established practices introduced by the Spring Framework. Spring Security tries to address all the layers of security inside your application. In addition, it comes packed with an extensive array of configuration options that make it very flexible and powerful.

Recall from the introduction in Chapter 1 that it can be said that Spring Security is simply a comprehensive authentication/authorization framework built on top of the Spring Framework. Although the majority of applications that use the framework are web based, Spring Security's core can also be used in standalone applications.

Many things make Spring Security immediately attractive to Java developers. To name just a few, I compiled the following list:

- **It's built on top of the successful Spring Framework** This is an important strength of Spring Security. The Spring Framework has become "the way" to build enterprise Java applications, and with good reason. It is built around good practices and two simple yet powerful concepts: dependency injection (DI) and Aspect-Oriented Programming (AOP). Also important is that a lot of developers have experience with Spring, so they can leverage that experience when introducing Spring Security in their projects.

- **It provides out-of-the-box support for many authentication models** Even more important than the previous point, Spring Security supports out-of-the-box integration with Lightweight Directory Access Protocol (LDAP), OpenID, Form authentication, Certificate X.509 authentication, database authentication, Jasypt cryptography, and lots more. All this support means that Spring Security adapts to your security needs—and not only that, it can change if your needs change, without much effort involved for the developer.

  This is important from a business point of view as well because the application can either adapt to the corporate authentication services or implement its own, thus requiring only straightforward configuration changes.

  This also means that there is a lot less software for you to write, because you are making use of a great amount of ready-to-use code that has been written and tested by a large and

active user community. You can, to a certain point, trust that this code works and use it with confidence. And if it does not work, you can always fix it and send a patch to those in charge of maintaining the project.

- **It offers layered security services** Spring Security allows you to secure your application at different levels, and to secure your web URLs, views, service methods, and domain model. You can pick and combine these features to achieve your security goals.

This is really flexible in practice. Imagine, for instance, that you offer services exposed through RMI, the Web, JMS, and others. You could secure all of these interfaces, but maybe it's better to secure just the business layer so that all requests are secured when they reach this layer. Also, maybe you don't care about securing individual business objects, so you can omit that module and use just the functionality you need.

- **It is open source software** Because it's part of the general SpringSource portfolio, Spring Security is an open source software tool. It also has a large community and user base dedicated to testing and improving the framework. Having the opportunity to work with open source software is an attractive feature for most developers. I know that the ability to look into the source code of the tools you like and work with is an exciting prospect. Whether our goal is to improve the tools or simply to understand how they work internally, we developers love to read code and learn from it.

Where Does Spring Security Fit In?

Spring Security is without question a powerful and versatile tool. But like anything else, it is not a tool that adapts to everything you want to do. Its offerings have a defined scope.

Where and why would you use Spring Security? Here is a list of reasons and scenarios:

- **Your application is in Java** The first thing to take into account is that Spring Security is written in the Java language and is a framework to be used only in Java applications (and other JVM languages as you will see in future chapters). So if you plan to work in a non-JVM language, Spring Security won't be of any use to you.
- **You need role-based authentication/authorization** This is the main use case of Spring Security. You have a list of users and a list of resources and operations on those resources. You group the users in roles and allow certain roles to access certain operations on certain resources. That's the core functionality.
- **You want to secure a web application from malicious users** I mentioned before that Spring Security is mostly used in web application environments. When this is the case, the first thing to do is allow only the users that you want to have access to your application, while forbidding the all others from even reaching it.
- **You need to integrate with OpenID, LDAP, Active Directory, and databases as security providers** If you need to integrate with a particular Users and Roles or Groups provider, you should take a look at the vast array of options Spring Security offers because integration might be already implemented for you, saving you from writing lots of unnecessary code. Sometimes you might not be exactly sure what provider your business will require to authenticate against. In this case, Spring Security makes your life easy by allowing you to switch between different providers in a painless way.
- **You need to secure your domain model and allow only certain users to access certain objects in your application** If you need fine-grained security (that is, you need to secure on a per object, per user basis), Spring Security offers the Access Control List (ACL) module, which will help you to do just that in a straightforward way.

- **You want a nonintrusive, declarative way for adding security around your application** Security is a cross-cutting concern, not really a core business functionality of your application (unless you work in a security provider firm). As such, it is better if it can be treated as a separate and modular add-on that you can declare, configure, and manage independently of your main business concerns. Spring Security is built with this in mind. By using Servlet Filters, XML configuration, and AOP concepts, the framework tries not to pollute your application with security rules. Even when using annotations, they are still metadata on top of your code. They don't mess with your code logic.
- **You want to secure your service layer the same way you secure your URLs, and you need to add rules at the method level for allowing or disallowing user access** Spring Security allows you to use a consistent security model throughout the layers of your application because it internally enforces this consistent model itself. You configure users, roles, and providers in just one place, and both the service and web layers make use of this centralized security configuration in a transparent way.
- **You need your application to remember its users on their next visit and allow them access** Sometimes you don't want or need the users of your application to log in every time they visit your site. Spring supports out-of-the-box, remember-me functionality so that a user can be automatically logged in on subsequent visits to your site, allowing them full or partial access to their profile's functionality.
- **You want to use public/private key certificates to authenticate against your application** Spring Security allows you to use X.509 certificates to verify the identity of the server. The server can also request a valid certificate from the client for establishing mutual authentication.
- **You need to hide elements in your web pages from certain users and show them to some others** View security is the first layer of security in a secured web application. It is normally not enough for guaranteeing security, but it is very important from a usability point of view because it allows the application to show or hide content depending on the user that is currently logged in to the system.
- **You need more flexibility than simple role-based authentication for your application** For example, suppose that you want to allow access only to users over 18 years of age using simple script expressions. Spring Security 3.1 uses the Spring Expression Language (SpEL) to allow you to customize access rules for your application.
- **You want your application to automatically handle HTTP status codes related to authorization errors (401, 403, and others)** The built-in exception-handling mechanism of Spring Security for web applications automatically translates the more common exceptions to their corresponding HTTP status codes—for example, `AccessDeniedException` gets translated to the `403` status code.
- **You want to configure your application to be used from other applications (not browsers) and allow these other applications to authenticate themselves against yours** Another application accessing your application should be forced to use authentication mechanisms in order to gain access. For example, you can expose your application through REST endpoints that other applications can access with HTTP security.
- **You are running an application outside a Java EE Server** If you are running your application in a simple web container like Apache Tomcat, you probably don't have support for the full Java EE security stack. Spring Security can be easily leveraged in these environments.

- **You are running an application inside a Java EE Server** Even if you are running a full Java EE container, Spring Security is arguably more complete, flexible, and easy to use than the Java EE counterpart.
- **You are already using Spring in your application and want to leverage your knowledge of it** I explained before some of the great advantages of Spring. If you are currently using Spring, you probably like it a lot. So you will probably like Spring Security as well.

Spring Security and Spring

As I said before, Spring Security is part of the SpringSource portfolio of open source projects. There are many more projects from SpringSource, and they are driven by a large and dynamic community of users. Among the mainstream SpringSource projects are the following:

- Spring Security
- Spring Batch
- Spring Integration
- Spring Web Services (WS)
- Spring Social
- Spring Web Flow
- Spring Data

All these projects are built on top of the facilities provided by the Spring Framework itself, which is the original project that started it all. You can think of Spring as the hub of all these satellite projects, providing them with a consistent programming model and a set of established practices. The main points you will see throughout the different projects is the use of DI, XML namespace-based configuration, and AOP, which as you will see in the next section, are the pillars upon which Spring is built on. In the later versions of Spring, annotations have become the most popular way to configure both DI and AOP concerns.

So Spring Security is just one more of these projects, and it is dedicated exclusively to addressing security concerns in your application.

If you read the online documentation, you will find out that Spring Security started originally as a non-Spring project. It was originally known as *The Acegi Security System for Spring*, and it was not the big and powerful framework it is today. Originally, it dealt only with authorization and leveraged container-provided authentication. Because of public demand, the project started to get traction, as more people started using it and contributing to its continuously growing code base. This eventually led to it becoming a Spring Framework portfolio project, and then later it was rebranded as "Spring Security."

So the project, for many years now, has been under the SpringSource umbrella of projects, powered by The Spring Framework itself.

But what exactly is the Spring Framework?

Spring Framework: A Quick Overview

I have already mentioned the Spring Framework project quite a lot. It makes sense to give an overview of it at this point, because many of the Spring Security characteristics I will cover in the rest of the book rely on the building blocks of Spring.

I admit I'm biased. I love Spring and have loved it for many years now. I think Spring has so many advantages and so many great things that I can't start a new Java project without using it. Additionally, I tend to carry its concepts around when working with other languages and look for a way to apply them because they now feel so natural to me.

There are many things that attract me to Spring, but the main ones are the two major building blocks of the framework: dependency injection (DI) and Aspect-Oriented Programming (AOP).

Why are these two concepts so important? Both of them are important because they allow you to develop loosely coupled, single-responsibility, DRY (Don't Repeat Yourself) code practically by default. These two concepts, and Spring itself, are covered extensively in other books and online tutorials; however, I'll give you a brief overview here.

Dependency Injection

The basic idea of DI, a type of Inversion of Control (IoC), is simply that instead of having an object instantiate its needed dependencies, the dependencies are somehow given to the object. In a polymorphic way, the objects that are given as dependencies to the target object that depends on them are known to this target object just by an abstraction (like an interface in Java) and not by the exact implementation of the dependency.

It's easier to look at this in code than explain it.

The object itself instantiates its dependencies (No dependency injection)

```
public class NonDiObject {

    private Helper helper ;

    public NonDiObject ( ) {
        helper = new HelperImpl ( ) ;
    }
    public void doStuffWithHelp( ) {
        helper.help( ) ;
    }
}
```

In this example, every instance of `NonDiObject` is responsible for instantiating its own `Helper` in the constructor. You can see that it instantiates a `HelperImpl`, creating a tight, unnecessary coupling to this particular `Helper` implementation.

The object receives its dependencies from some external source (with dependency injection)

```
public class DiObject {

    private Helper helper ;

    public DiObject(Helper helper) {
        this.helper = helper;
    }
    public void doStuffWithHelp( ) {
        helper.help( ) ;
    }
}
```

In this version, the `Helper` is passed to the `DiObject` at construction time. `DiObject` is not required to instantiate any dependency. It doesn't even need to know how to do that or what particular implementation type the `Helper` is, or where it comes from. It just needs a helper and uses it for whatever requirement it has.

The advantage of this approach should be clear. The second version is loosely coupled to the `Helper`, depending only on the `Helper` interface, allowing the concrete implementation to be decided at runtime and thus giving lots of flexibility to the design.

Spring dependency injection configuration is normally defined in XML files, although later versions have turned more to annotation-based configuration and Java-based configuration.

Aspect Oriented Programming (AOP)

AOP is a technique for extracting cross-cutting concerns from the main application code and applying them in a transverse way across the points where they are needed. Typical examples of AOP concerns are transactions, logging, and security.

The main idea is that you decouple the main business logic of your application from special-purpose concerns that are peripheral to this core logic, and then apply this functionality in a transparent, unobtrusive way through your application. By encapsulating this functionality (which is simply general application logic and not core business logic) in its own modules, they can be used by many parts of the application that need them, avoiding the need to duplicate this code all over the place. The entities that encapsulate this cross-cutting logic are referred to as Aspects in AOP terms.

There are many implementations of AOP in Java. The most popular, perhaps, is AspectJ which requires a special compilation process. Spring supports AspectJ, but it also includes its own AOP implementation, known simply as *Spring AOP*, which is a pure Java implementation that requires no special compilation process.

Spring AOP using proxies is available only at the public-method level and just when it is called from outside the proxied object. This makes sense because calling a method from inside the object won't call the proxy; instead, it calls the real self object directly (basically a call on the `this` object). This is something very important to be aware of when working with Spring, and sometimes it is overlooked by novice Spring developers.

Even when using its own AOP implementation, Spring leverages the AspectJ syntax and concepts for defining Aspects.

Spring AOP is a fairly big subject, but the principle behind the way it works is not difficult to understand. Spring AOP works with the use of dynamically created proxy objects that take care of the AOP concerns around the invocation of your main business objects. You can think of the proxy and Spring AOP in general simply as a Decorator Pattern implementation, where your business object is the component and the AOP proxy is the decorator. Figure 2-1 shows a simple graphical representation of the concept. Thinking about it this way, you should be able to understand Spring AOP easily. The following code shows how the magic happens conceptually.

Our business object, not transactional

```
public class Business Object implements BusinessThing {

    public void doBusinessThing( ) {
        // Some business stuff
    }
}
```

Suppose you have an aspect for transactions. Spring creates dynamically at runtime an object that conceptually looks like the following code.

Spring AOP magic

```
public class BusinessObjectTransactionalDecorator implements BusinessThing {

    private BusinessThing component ;

    public BusinessObjectTransactionalDecorator(BusinessThing component ) {
        this . component = component ;
    }
```

```
    public void doBusinessThing( ) {
        // some start transaction code
           component.doBusinessThing( ) ;
        // some commit transaction code
    }
}
```

Again, remember this simple idea and Spring AOP should be easier to understand.

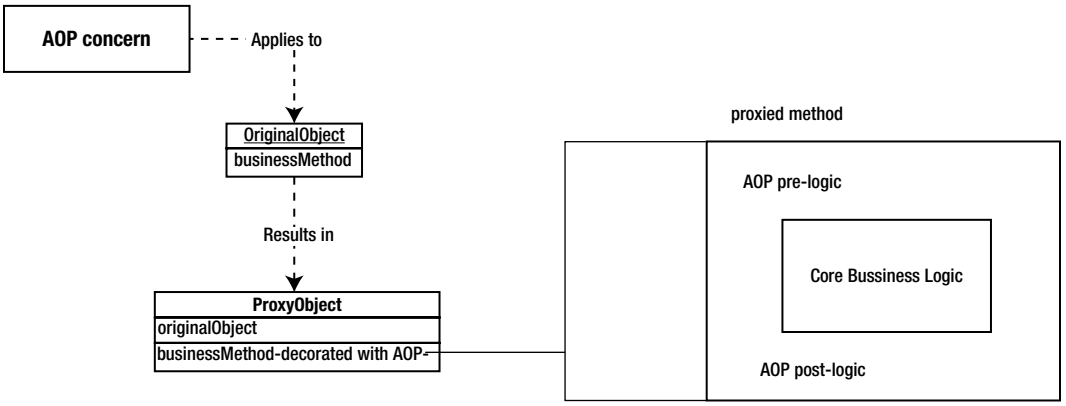


Figure 2-1. *Spring AOP in action*

An Initial Spring Security Secured Application

Spring Security builds upon the concepts defined in the previous section and integrates nicely into the general Spring ecosystem. You need to understand those concepts well to take maximum advantage of Spring Security. However, you could start using Spring Security without really knowing all these details, and then learn them as you progress and look to do more advanced things.

As many other programming books do, I will show an initial example of the subject with the familiar "Hello World" application. It will need a tweak to make sense for our purposes. So I'll show the "Hello World" message just to authorized users and not allow unauthorized users to see the message.

In this example, we will create a simple web project powered by Spring Security. It will be a simple Servlet-based web application.

We will use Maven 3.0.4 throughout the book to build our projects as well as Java 1.7. To make sure that the examples in the book work as expected, try to use the same versions. All examples have been tested in both Linux (Ubuntu 12.04) and Mac OS X 10.7.4.

Let's create a simple Maven web project. From the command line, go to any folder you want to use as the root of your projects and execute the following:

```
mvn archetype:generate -DgroupId=com.apress.pss -DartifactId=pss01 -DarchetypeArtifactId=maven-
archetype-webapp
```

Now add the Servlet dependency to the new project. To do that, make the pom.xml file look like Listing 2-1.

Listing 2-1. The pom.xml file with Servlet dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.pss</groupId>
    <artifactId>pss01</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>pss01 Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.5</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
    <build>
 <plugins>
  <plugin>
  <groupId>org.mortbay.jetty</groupId>
  <artifactId>jetty-maven-plugin</artifactId>
  <version>8.1.1.v20120215</version>
  </plugin>
 </plugins>
 <finalName>pss01</finalName>
 </build>
</project>
```

As you can see from the pom.xml listing, I included the Jetty plugin dependency. I will be working throughout the book mostly with the Jetty server because it embeds nicely in the Maven life cycle. Using any other container (like Apache Tomcat) to run the code from the book should be no problem, and it should work without issues.

You have now created a standard Maven web application. The next task is to create a simple Servlet that will render the "Hello World" message. Create the following Servlet in the `com.apress.pss.servlets` package. See Listing 2-2.

■ **Note** Remember that you have to keep the conventional Maven file structure. This means you have to create the `/src/main/java` folders for your source classes if you don't have them yet. Look at the Maven documentation online at `http://maven.apache.org/` for more details.

Listing 2-2. HelloWorldServlet

```
package com.apress.pss.servlets;
import java.io.IOException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns ={"/hello" } )
public class HelloWorldServlet extends HttpServlet {

    private static final long serialVersionUID = 2218168052197231866L ;

    @Override
    public void doGet (HttpServletRequest request , HttpServletResponse response){
        try {
              response.getWriter( ).write( "Hello World" ) ;
        } catch(IOException e) {
              e.printStackTrace( ) ;
        }
    }
}
```

In the command line, in the root of the project (where the pom.xml file resides), execute the following:

```
mvn jetty : run
```

Then visit the URL `http://localhost:8080/hello` in the browser. You'll see the expected "Hello World" message, as shown in Figure 2-2.

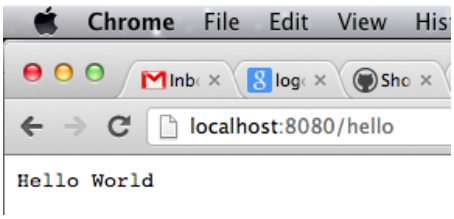


Figure 2-2. *The "Hello World" message*

Now we want to secure this page and allow only authenticated users to be able to read the super top-secret "Hello World" message. To do this, we will use Spring Security.

Download from Wow! eBook <www.wowebook.com>

Let's say that only the Scarvarez family (whose members are Carl, Mon, Bea, and Andr) is allowed to see this "Hello World" message. The rest of the world is not allowed to. To make this work, you need to do the following:

1. Import the required Spring Framework and Spring Security libraries into the project.
2. Configure the project to be aware of Spring Security.
3. Configure the users and roles that will be part of the system.
4. Configure the URLs that you want to secure.

I'll give more detail about each of these steps in the next four sections.

Adding Spring Security (and Spring Core Itself) to the Project

In this section, we'll start our journey into the inner workings of the framework and see its main building blocks and how it works. The information will be mostly introductory. I'll conduct a full, in-depth review of the framework in the next chapter. I could just tell you what to add to the project to make the application work, but I think it's better to tell you first what the different components of the framework are so that you can start coding with a better knowledge of how the framework is built. This means that I will tell you how to grab the source code of the project and build it, and then explain in a general way the different modules that make up the framework.

Spring Security Source

Open source software has an invaluable characteristic for software developers: free access to all its source code. With this, we can understand how our favorite tools and frameworks work internally, and we also can learn a lot about the way other (perhaps very good) developers work, including what practices, techniques, and patterns they use. Free access to source code also enables us, in general, to gather ideas and experience for our own development. Not only that; as a more practical matter, having access to the source code allows us to debug these applications in the context of our application: we can find bugs or simply follow our application's execution through them.

That is the way I work many times, anyway. I really like understanding the frameworks I work with and learning from the great work and effort put into them by some very talented developers around the world.

A lot of this book's content is based on looking at the source code of Spring Security. Where appropriate, some of that code will be printed in the pages of this book to make points more clear.

So let's grab the Spring Security source code.

Currently, Spring Security and most Spring projects live in Github. You probably know about Github (`https://github.com/`). However, if you don't, you should definitely take a look at it because it has become a standard public source-code repository for many open source projects in a multitude of programming languages.

Github (`https://github.com/`) is a repository, and a hosting service for Git repositories, with a very friendly management interface. The Spring Security project can be found inside the SpringSource general Github section at `https://github.com/SpringSource/spring-security`. To get the code, go to any location you want to have the project in your command-line terminal and execute the following command:

```
git clone https://github.com/SpringSource/spring-security.git
```

That's it, now you have the Git repository cloned in your computer. Let's go inside the newly created directory and see what is there. (See Figure 2-3.)

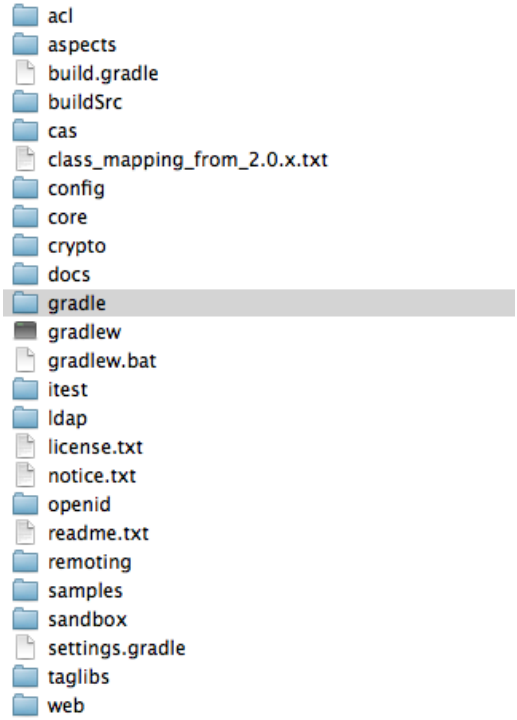


Figure 2-3. *Spring Security source code folder*

There are a lot of files and directories here. The first thing that is worth mentioning is that the current versions of Spring Security use Gradle as the build tool.

Until version 2.5.0.M1, Spring Security used Maven as the build tool. Starting with version 3.0.0.RC2, Gradle was also included. Maven support was finally removed in version 3.1.0.M1, leaving only Gradle as the build system.

Gradle is a build tool written in Groovy that uses an internal Domain Specific Language to specify the build configuration of your project. A detailed explanation of Gradle is outside the scope of this book, so I will just cover what you need to build Spring Security from the source code.

You can see that in the top directory there is a file named gradlew. This is what is known as a *Gradle Wrapper*, and its main advantage is that it is a self-contained build script-tool. When you run it, it will download Gradle for you and use the downloaded Gradle to build your project.

Go ahead and run `gradlew build install` from the top directory and wait for the project to build.

■ **Note** It is possible for the build to fail in your system. Because we are working in the master branch of the framework, a lot of code is being committed there all the time. It is not under my control if this build fails at any given time. Later, you will simply use a stable tag for developing the examples in the book.

You should now have the project built and available in your current Maven repository. (Yes, when using Gradle, the dependencies can be simply stored in the Maven repository.) You can see that it is very straightforward to build the project from the source code.

Setting the project build aside for now, take another look at the contents of the top project directory. Most of the folders in the directory correspond to individual subprojects or modules that break the functionality of Spring Security into more discrete and specialized units.

Spring Security currently comprises several modules:

- **`core`** As its name indicates, this is the main module in the project. Every other module builds on top of this one because it provides core functionality to the rest of the framework. In this module, you can find the main interfaces and classes that establish the concepts and hook points that the rest of the system uses. It also contains the core implementations of the system, including the JDBC authentication support, the Java Authentication and Authorization Service (JAAS) authentication provider, the access voting system, MD5 and SHA password encoders, and a lot more.
- **`web`** This module deals with the web-layer security of your application. It builds on the core and leverages its main abstractions and implementations to provide security to your web-based application. In this module, you can find the Servlet Filters that deal with the pre-processing and post-processing of Servlet requests, the Servlet Session management, the Secure Sockets Layer (SSL) support, the Remember Me support, the HTTP status code generation, and more web-related things.
- **`config`** This module contains the namespace definition, the XSDs, and the validation and parsing rules of the different elements of the framework. When you write your configuration files for your application using the XML definitions for the different Spring Security configuration options, this module is responsible for parsing that XML and creating standard bean definitions from them, wiring them together, and getting them ready to be instantiated by the Spring context-loading process. Here, you will find the translation from namespaced xml elements to standard Spring Framework `<bean>` elements.
- **`taglibs`** This module contains the `taglib` definitions and implementations for configuring security at the JSP level, allowing the view content to be conditionally rendered depending on the security constraints that you chose to define in the tags.
- **`acl`** This module contains all the logic needed for the access control list (ACL) security model available with Spring Security. It contains the database model, including the SQL DDLs (Data Definition Language), to establish the rules and relationships between domain entities and their security restrictions. I will cover ACLs in depth in the chapter dealing with business-layer security.
- **`remoting`** Leverages security support for remoting protocols. Currently supported is the Spring `HttpInvoker`, which uses BASIC authentication, and Java RMI. This module takes care of propagating the security context to these remote services.
- **`ldap`** Contains all the functionality required to connect to and authenticate against an LDAP service, including the `AuthenticationProvider` and `UserDetails` LDAP implementations. It also includes an embeddable ApacheDS server implementation for testing and development purposes.
- **`cas`** This is the Spring Security support for the JA-SIG Central Authentication Service (CAS) single sign-on service. So if you want to authenticate with CAS, you need this module in your application.
- **`openid`** Again, this module is well described by its name. It contains OpenID support, including the OpenID-specific `AuthenticationProvider`, and it leverages the `openid4java` library `http://code.google.com/p/openid4java/`.

- **aspects** Contains an AspectJ Aspect that cuts into the execution of methods with various security annotations (`@Secured`, `@PreAuthorize`). It delegates to a configured Security Interceptor.
- **crypto** Provides various encryption, ciphering, and codec implementations, as well as some password-encoder strategies, to use in your application. There is also a *samples* folder inside the top directory. Here, you can find some example applications to help you get to know the framework and how to configure your applications.

Finally, you need to add security to the application. For our project example, you need to depend on the core, web, and config modules. So we add them to our pom.xml, as shown in Listing 2-3.

Listing 2-3. Spring Security Maven dependencies

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1.1</version>
</dependency>
```

■ **Note** As you probably know, Maven handles transitive dependencies. In our example, the dependencies we just defined in Listing 2-3 will translate into more real-world dependencies, including the needed dependencies from the Spring Framework. This includes Spring web support and the core libraries for DI and `ApplicationContext` management.

Configuring the Web Project To Be Aware of Spring Security

To activate Spring Security in a Java web application, you need to configure a particular Servlet filter that will take care of pre-processing and post-processing the requests, as well as managing the required security constraints. Make your web.xml look like Listing 2-4.

Listing 2-4. The web.xml with Spring Security integration

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

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

You are specifying here that requests to all URLs (`/*`) will go through the `DelegatingFilterProxy` filter. The name of this filter is important because it is the default name Spring Security will use to configure its filter chain. I will explain all this in detail later. For now, let's continue with the example.

You now need to define the actual Spring Security configuration. For that, create a file named `applicationContext-security.xml` in the `WEB-INF` folder and put the content shown in Listing 2-5 into it:

Listing 2-5. A Spring application context with the security configuration applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true">
        <security:intercept-url pattern="/hello" access="ROLE_SCARVAREZ_MEMBER" />
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="car" password="scarvarez" />
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="mon" password="scarvarez" />
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="bea" password="scarvarez" />
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="andr" password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans>
```

That's all we need to secure the given URL for the Scarvarez family only. You can see that you are defining a role called `"ROLE SCARVAREZ MEMBER"` that is the only one that has access to the url `"/hello"`. You also see that you are assigning this role to all the Scarvarez family members.

To make this work, you need to reference this new file in the web.xml. So open it again and add the content from Listing 2-6 to it.

Listing 2-6. Listener configuration for loading the Spring application context. It should be added at the top of the file after the <web-app> element. Look at the Servlet specification for details

```
<listener>
  <listener-class>
     org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
       /WEB-INF/applicationContext-security.xml
    </param-value>
</context-param>
```

That's all the configuration needed at the moment. Go to the command line, execute `mvn jetty:run` again, point your browser to the URL `http://localhost:8080/hello`, and press the Enter key. What do you see?

You should see a login page asking you to provide the user name and password as Figure 2-4 shows. Go ahead, impersonate a Scarvarez family member, and try to log in. For example, use **car** as the user name and **scarvarez** as the password. You should now be allowed access to the same "Hello World" page as Figure 2-2 showed before. If you instead tried to log in as a nonexistent user, you should get the page shown in Figure 2-5.

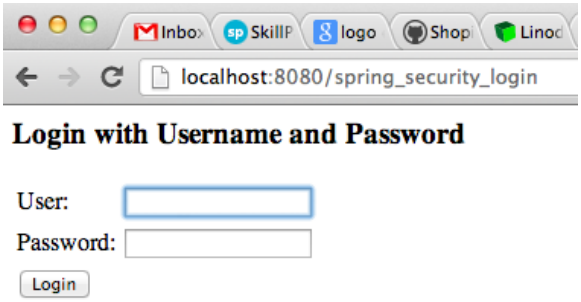


Figure 2-4. *Login page*

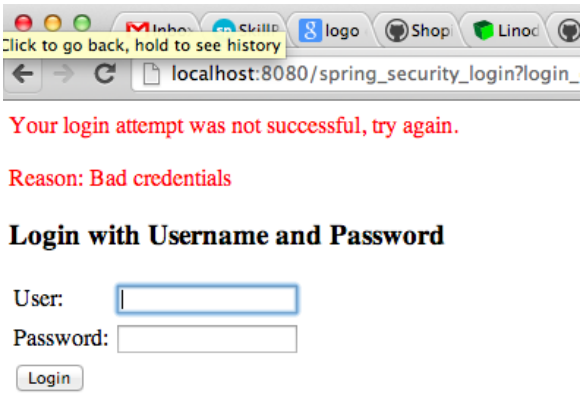


Figure 2-5. *The result of trying to log in with an incorrect user name and password combination*

This all seems pretty cool for so little work. As a matter of fact, we haven't written any code; we just did some XML configuration. Now our Servlet-based application is secured to allow access only for authenticated users. The fact that we are using this XML configuration to secure the application is very relevant, because it shows that we are adding security in a completely unobtrusive and declarative way to an application that is otherwise insecure.

This is all well and nice, but right now it looks like magic. Where does this login form come from? How does this `<http auto-config>` tag secure the URLs?.

In the next section, we will cover the basics of what is going on. A full, in-depth explanation of Spring Security follows in the next chapter.

Understanding the Simple Application

You can see that with some relatively simple configuration, we managed to secure our regular Java Web application with a functional role-based authentication/authorization scheme. I can imagine an application for which this simple configuration would be enough security—for example, an application that has an admin section and an open-to-all section, and where the admin user (or users) is predefined beforehand.

Even for this simple-looking configuration, there is a lot going on under the hood of your application, but Spring Security takes care of the heavy work and leaves you to worry about what you need to worry about:—establishing your security rules. But what is happening here exactly?

What follows is a general overview of the process, followed by a graphical representation of it:

1. Start the application (in this case, by using `jetty run`). When the application is loading, it looks at its web.xml file. There it finds your Listener declaration `ContextLoaderListener`. This listener uses the path defined in the context `param contextConfigLocation` to find the Spring xml file that it needs to load.

2. The `applicationContext-security.xml` file is loaded and parsed by Spring. This file uses the custom Spring Security namespace to define, in a friendly way, the different security concerns that we need to address.

3. The supporting beans for using Spring Security are instantiated and initialized by the `Context LoaderListener` after parsing the XML in the normal Spring way.

4. Your web.xml filter (of class `DelegatingFilterProxy`) is instantiated and added to the filter chain of your web application. This Servlet filter implementation is a Spring context-aware filter that simply delegates the filtering to Spring-defined filter beans. In this case, it will delegate to a bean named `springSecurityFilterChain`, which is defined implicitly by your use of the Spring Security XML namespace. At this point, the application is initialized.

5. You make a request to the URL `http://localhost:8080/hello`. The request gets to the Jetty server, Jetty wraps it into an `HttpServletRequest`, and sends it through to the filter `springSecurityFilterChain`. This filter is a composite of numerous filters that deal with different parts of the authentication/authorization process.

6. Somewhere in the filter chain, Spring notices that you are trying to access a URL that needs the special_role `"ROLE_SCARVAREZ_MEMBER"` to be able to continue. Spring looks at the current credentials in the web session and can't find any user details with this role.

7. When Spring realizes (inside one of the filters) that there is no current user in the session with permission to access the requested URL, it generates a redirect response that redirects the user browser to a login URL that contains a self-generated login form.

8. The new form is rendered. When the form is filled out and submitted, the request is sent to a special URL`"/j_spring_security_check"` in the current domain, which in our case is `http://localhost:8080`. When Spring Security detects that this URL has been requested, it tries to extract the user name and password from the request and creates an authentication request.

9. The authentication request is sent to an authentication manager, which you configured in your XML file. Internally, the manager finds the user (if it exists) and fills one authentication object with the user details, credentials, and authorities. This is now an authenticated user, and a session is associated with this user. If the details contained in the authentication don't match any user, an exception is thrown indicating that. This exception is then handled by a different flow that takes care of presenting the login screen again with the error message.

10. A redirect response is created to the original requested URL.

11. When this new redirect request gets in the system, it goes through the filters again, but this time the authentication ones are not called because there is already an authenticated user. This time, a different filter will get called. This filter compares the authenticated user's authorities against the authorities needed to have in order to access the requested resource (in this case, the `/hello` URL).

12. The framework decides that the user is allowed to access the resource, so the filter chain finally forwards the request to the Servlet. You see the "Hello World" message. This is the happy-path scenario of how the flow works. If anything goes wrong (for example, if the user is not found), Spring Security normally throws an exception and, in the case of web security, maps that exception to a relevant HTTP status code. This will be covered later in the book as we go deeper into the work of the framework.

Figure 2-6 shows all this interaction.

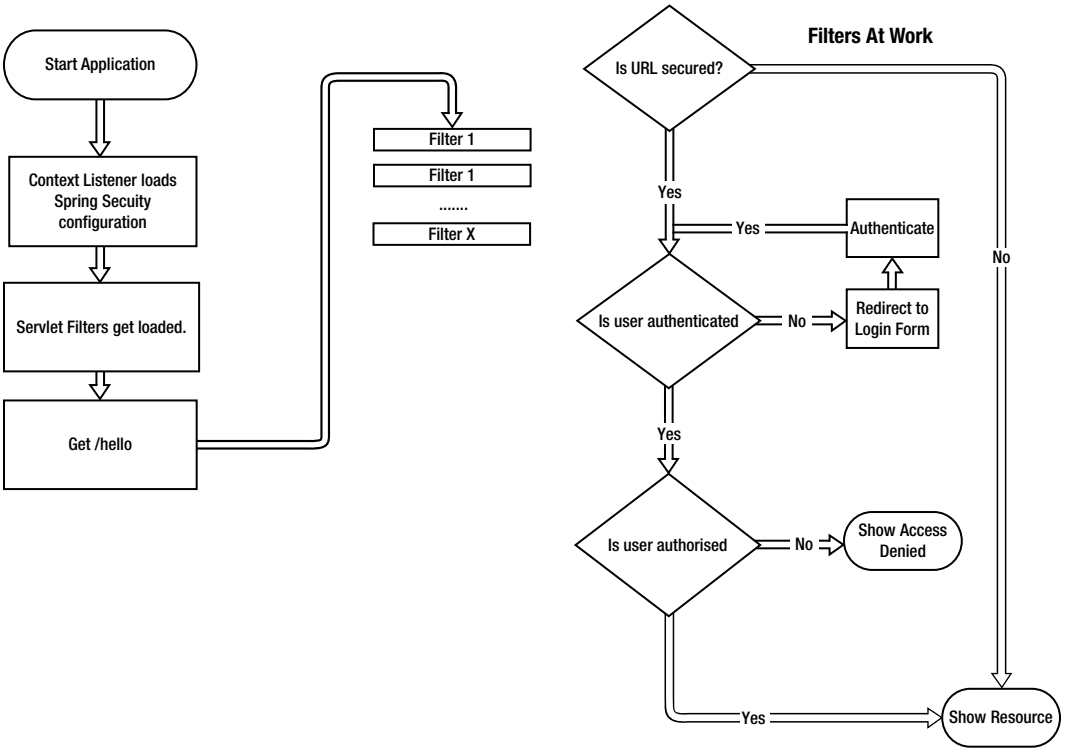


Figure 2-6. *Spring Security from load up to first request*

Wow! That is a lot of work that Spring Security is doing on our behalf while we just had to define some simple XML configuration. And this is only the tip of the iceberg. In the next chapter, I will dive deeply into how all this works internally when we look at the architecture of Spring Security.

Summary

Right now, you should have a good idea of what Spring Security is and what it is useful for. You built your first Spring Security–powered application. Along the way, I introduced some of the major architectural and design principles behind it and how they are layered on top of the great Spring Framework. I also gave you a quick look at the source code of the Spring project and at the different modules that make up the framework. I introduced dependency injection and AOP and gave an overview of the step-by-step process a typical web application goes through when it is secured with Spring Security. In the next chapter, I'll go deep into the architecture and design of the framework.

CHAPTER 3

Spring Security Architecture and Design

In the previous chapter, I developed an initial application secured with Spring Security. I gave an overview of the way this application worked and looked in detail at some of the Spring Security components that are put into action in common Spring Security–secured application. In this chapter, I am going to extend those explanations and delve deeply into the framework.

I'll look at the main components of the framework, explain the work of the servlet filters for securing web applications, look at how Spring AOP (Aspect Oriented Programming) helps you add security in an unobtrusive way, and in general, show how the framework is designed internally.

What Components Make Up Spring Security?

In this section, I'll take a look at the major components that make Spring Security work. I'll offer a big-picture overview of the framework and then delve deeper into each major component.

The 10,000-Foot View

Spring Security is a relatively complex framework that aims to make it easy for the developer to implement security in an application. At the most general level, it's a framework composed of intercepting rules for granting, or not granting, access to resources. Figure 3-1 illustrates this.

Download from Wow! eBook <www.wowebook.com>

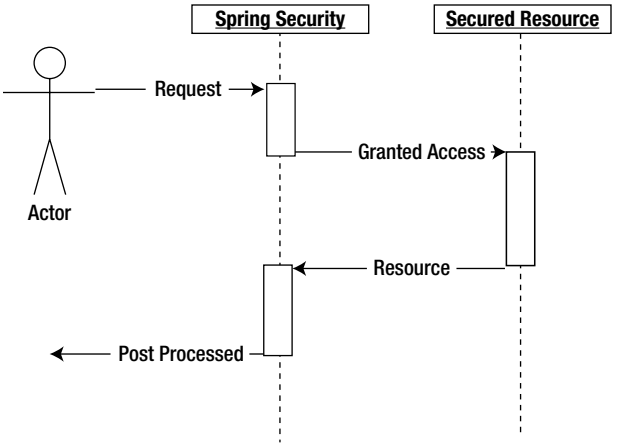


Figure 3-1. *Spring Security 10,000-foot overview*

From this view, you can think of Spring Security simply as an extra layer built on top of your application, wrapping specific entry points into your logic with determined security rules.

The 1,000-Foot View

Going into a little more detail, we arrive at AOP and servlet filters.

Spring Security's interception model of security applies to two main areas of your application: URLs and method invocations. Spring Security wraps around these two *entry points* of your application and allows access only when the security constraints are satisfied. Both the method call and the filter-based security depend on a central *Security Interceptor*, where the main logic resides to make the decision whether or not access should be granted.

In Figure 3-2, you can see this more detailed overview of the framework.

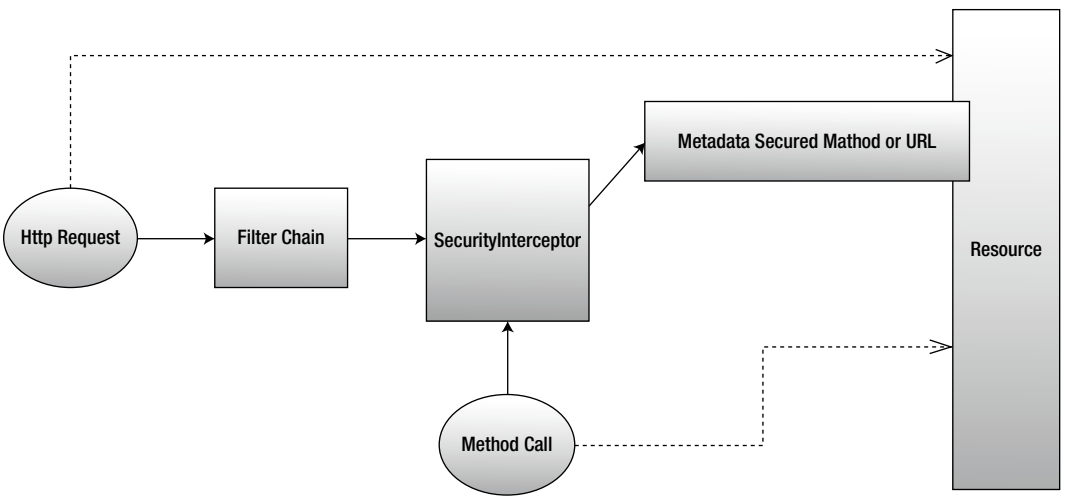


Figure 3-2. *In this view, both method calls and HTTP requests try to access a resource, but first they must go through SecurityInterceptor*

The 100-Foot View

Spring Security might seem simple conceptually, but internally there is a lot going on—in a very well-built software tool. This next overview will show you the main collaborating parts that participate in the general process of ensuring that your security constraints are enforced. This is particularly achievable with an open source project like Spring Security which allows you to get into the framework itself and appreciate its design and architecture by accessing directly the source code.. After that, I'll delve deeper into the implementation details.

For me, what follows is the best way to understand Spring Security from the inside. The enumeration of what I consider to be the main components of the framework will help you know where everything belongs and how your application is enforcing the security rules that you specify for it. Figure 3-3 provides an illustration.

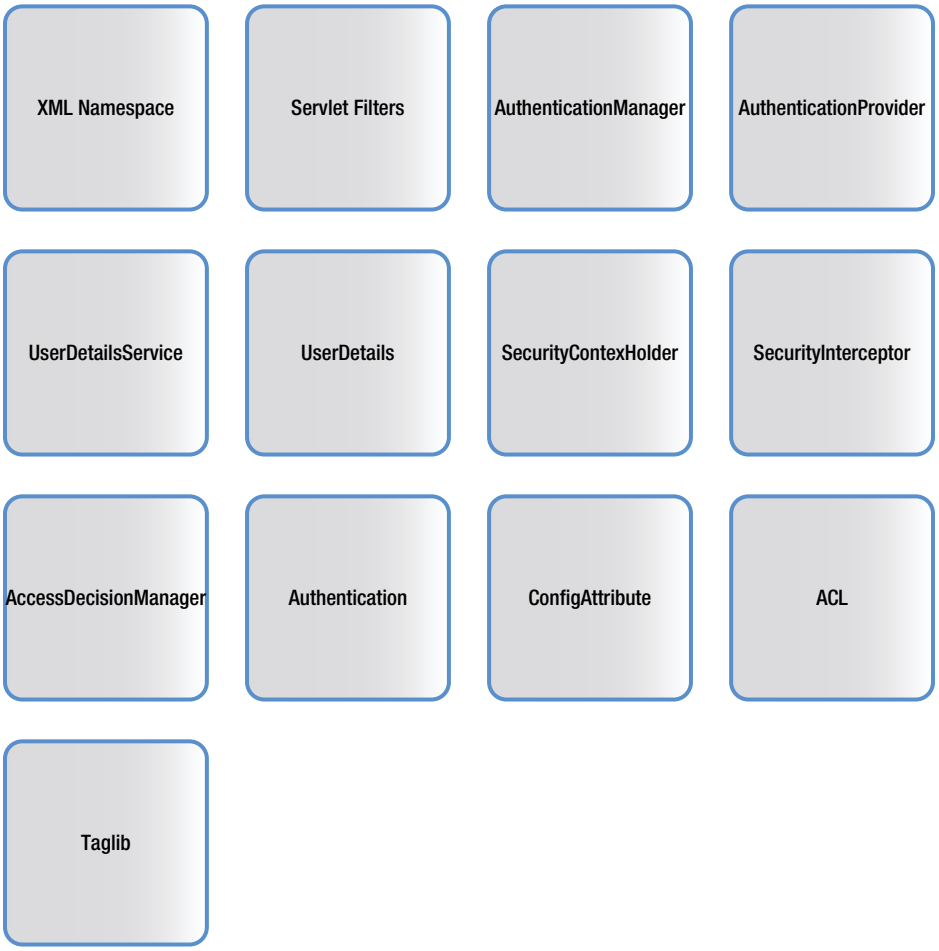


Figure 3-3. *The key components of Spring Security*

The Security Interceptor

One of the most important components of the framework is the Security Interceptor. With the main logic implemented in `AbstractSecurityInterceptor` and with two concrete implementations in the form of `FilterSecurityInterceptor` and `MethodSecurityInterceptor` (as shown in Figure 3-4), the Security Interceptor is in charge of deciding whether a particular petition should be allowed to go through to a secured resource. `MethodSecurityInterceptor`, as its name should tell you, deals with petitions directed as method calls, while `FilterSecurityInterceptor` deals with petitions directed to web URLs.

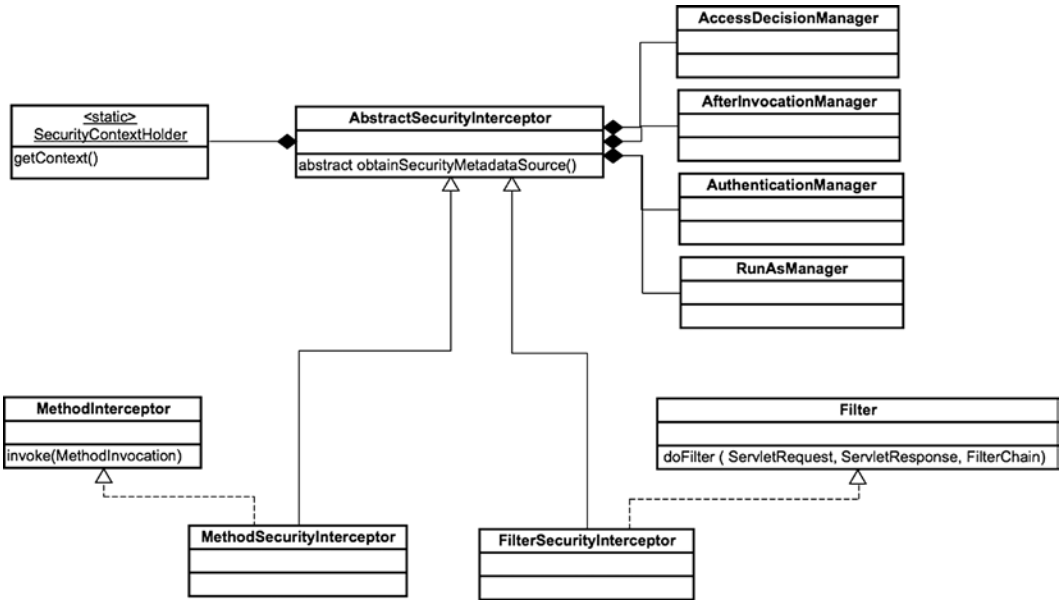


Figure 3-4. *SecurityInterceptor UML class diagram simplified*

The Security Interceptor works with a preprocessing step and a postprocessing step. In the preprocessing step, it looks to see whether the requested resource is secured with some metadata information (or `ConfigAttribute`). If it is not, the request is allowed to continue its way either to the requested URL or method. If the requested resource is secured, the Security Interceptor retrieves the `Authentication` object from the current `SecurityContext`. If necessary, the `Authentication` object will be authenticated against the configured `AuthenticationManager`. After the object is authenticated, `AccessDecisionManager` is called to determine if the authenticated entity is able to finally access the resource. `AccessDecisionManager` throws an `AccessDeniedException` if the authenticated entity is not allowed to access the resource. If `AccessDecisionManager` decides that the `Authentication` object is allowed to access the resource, the `Authentication` object is passed to `RunAsManager` if this is configured. If `RunAsManager` is not configured, a no-op implementation is called. `RunAsManager` returns either null (if it's not configured to be used) or a new `Authentication` object containing the same principal, credentials, and granted authorities as the original `Authentication` object, plus a new set of authorities based on the RUN_AS that is being used. This new `Authentication` object is put into the current `SecurityContext`.

After this processing, and independently of whether or not a run-as `Authentication` object is used, the Security Interceptor creates a new `InterceptorStatusToken` with information about the `SecurityContext` and the `ConfigAttributes`. This token will be used later in the postprocessing step of the Security Interceptor. At this point, the Security Interceptor is ready to allow access to the secured resource, so it passes the invocation through and the

particular secured entity (either a URL or a method) is invoked. After the invocation returns, the second phase of the Security Interceptor comes into play, and the postprocessing begins. The postprocessing step is considerably simpler, and it involves only calling a `AfterInvocationManager`'s decide method if there is one configured. In its current implementation `AfterInvocationManager` delegates to instances of `PostInvocationAuthorizationAdvice`, which ultimately can filter the returned objects or throw a `AccessDeniedException` if necessary. This is the case if you are using the postinvocation filters in method-level security, as you will see in the following chapter. In the case of web security, the `AfterInvocationManager` is null.

That is a lot of work for the Security Interceptor. However, because the framework is nicely modular at the class level, you can see the Security Interceptor simply delegates most of the task to a series of well-defined collaborators, which in a very SRP (Single Responsibility Principle) way focus on single, narrowly scoped responsibilities. This is good software design and an example you should emulate. As shown in Listing 3-1, I paste the main parts of the code from the `AbstractSecurityInterceptor` itself so that you can see the things I've been talking about. I include some comments in the code so that you can understand better what it does. My comments start with // ----.

Listing 3-1. AbstractSecurityInterceptor

```
protected InterceptorStatusToken beforeInvocation(Object object) {
    Assert.notNull(object, "Object was null");
    final boolean debug = logger.isDebugEnabled();

    //  Here we are checking if this filter is able to process a
    // particular type of object.
    // For example FilterSecurityInterceptor is able to process
    // FilterInvocation objects.
    // MethodSecurityInterceptor is able to process MethodInvocation
    // objects.

    if (!getSecureObjectClass().isAssignableFrom(object.getClass())) {
        throw new IllegalArgumentException("Security invocation attempted" +
                " for object " + object.getClass().getName() + " but " +
                "AbstractSecurityInterceptor only configured to support" +
                " secure objects of type:" + getSecureObjectClass());
    }

    // Here we are retrieving the security metadata that maps to the
    // object we are receiving.
    // So if we are receiving a FilterInvocation,
    // the request is extracted from it and used to find the
    // ConfigAttribute (s) that match the request path pattern

    Collection<ConfigAttribute> attributes = this
            .obtainSecurityMetadataSource().getAttributes(object);
    if (attributes == null || attributes.isEmpty()) {
        if (rejectPublicInvocations) {
            throw new IllegalArgumentException("Secure object invocation " +
                    "" + object +
                    " was denied as public invocations are not allowed " +
                    "via this interceptor. " + "This indicates a " +
                    "configuration error because the " +
                    "rejectPublicInvocations property is set to 'true'");
        }
```

```
        if (debug) {
            logger.debug("Public object - authentication not attempted");
        }
        publishEvent(new PublicInvocationEvent(object));
        return null; // no further work post-invocation
    }
    if (debug) {
        logger.debug("Secure object: " + object + "; Attributes: " +
                attributes);
    }
    if (SecurityContextHolder.getContext().getAuthentication() == null) {
        credentialsNotFound(messages.getMessage
                ("AbstractSecurityInterceptor.authenticationNotFound",
                        "An Authentication object was not found in the "
                                + "SecurityContext"), object, attributes);
    }
    Authentication authenticated = authenticateIfRequired();

    // Here we are calling the decision manager to decide if
    // authorization is granted or not.

    // This will trigger the voting mechanism,
    // and in case that access is
    // not granted an exception
    // should be thrown.

    try {
        this.accessDecisionManager.decide(authenticated, object,
                attributes);
    } catch (AccessDeniedException accessDeniedException) {
        publishEvent(new AuthorizationFailureEvent(object, attributes,
                authenticated, accessDeniedException));
        throw accessDeniedException;
    }
    if (debug) {
        logger.debug("Authorization successful");
    }
    if (publishAuthorizationSuccess) {
        publishEvent(new AuthorizedEvent(object, attributes,
                authenticated));
    }
    //  Here it will try to use the run-as functionality of Spring
    // Security that allows a user
    // to impersonate another one acquiring its security roles,
    // or more precisely, its
    // GrantedAuthority (s)

    Authentication runAs = this.runAsManager.buildRunAs(authenticated,
            object, attributes);
```

```
    if (runAs == null) {
        if (debug) {
            logger.debug("RunAsManager did not change Authentication " +
                    "object");
        }
        // no further work post-invocation
        return new InterceptorStatusToken(SecurityContextHolder
                .getContext(), false, attributes, object);
    } else {
        if (debug) {
            logger.debug("Switching to RunAs Authentication: " + runAs);
        }
        SecurityContext origCtx = SecurityContextHolder.getContext();
        SecurityContextHolder.setContext(SecurityContextHolder
                .createEmptyContext());
        SecurityContextHolder.getContext().setAuthentication(runAs);
        // need to revert to token.Authenticated post-invocation
        return new InterceptorStatusToken(origCtx, true, attributes,
                object);
    }
    //  If the method has not thrown an exception at this point,
    // it is safe to continue
    // the invocation through to the resource. Authorization has
    // been granted.
}

protected Object afterInvocation(InterceptorStatusToken token,
                                 Object returnedObject) {
    if (token == null) {
        // public object
        return returnedObject;
    }
    if (token.isContextHolderRefreshRequired()) {
        if (logger.isDebugEnabled()) {
            logger.debug("Reverting to original Authentication: " + token
                    .getSecurityContext().getAuthentication());
        }
        SecurityContextHolder.setContext(token.getSecurityContext());
    }
    //  If there is an afterInvocationManager configured,
    // it will be called.
    // It will take care of filtering the return value or actually
    // throwing an exception
    // if it is relevant to do so.
    if (afterInvocationManager != null) {
        // Attempt after invocation handling
        try {
            returnedObject = afterInvocationManager.decide(token
                    .getSecurityContext().getAuthentication(),
                    token.getSecureObject(), token.getAttributes(),
                    returnedObject);
```

```
        } catch (AccessDeniedException accessDeniedException) {
            AuthorizationFailureEvent event = new AuthorizationFailureEvent(
                    token.getSecureObject(), token.getAttributes(),
                    token.getSecurityContext().getAuthentication(),
                    accessDeniedException);
            publishEvent(event);
            throw accessDeniedException;
        }
    }
    // Here is the full authorization cycled finished.
    // The response is returned to the caller.
    return returnedObject;
}
```

The Security Interceptor lies at the core of the Spring Security framework. Every call to a secured resource in Spring Security passes through this interceptor. The `AbstractSecurityInterceptor` shows its versatility when you realize that two not very related kinds of resources (URL endpoints and methods) leverage more of the functionality of this abstract interceptor. This, once again, sh-ows the effort put into the design and implementation of the framework.

Figure 3-4 shows the interceptor in a UML (Unified Modeling Language) class diagram. And Figure 3-5 shows a simplified sequence diagram.

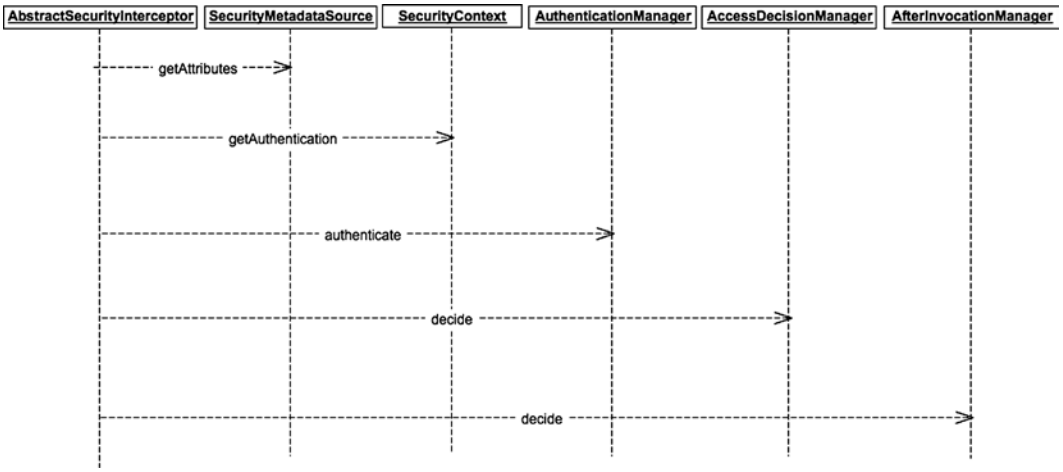


Figure 3-5. *AbstractSecurityInterceptor sequence diagram simplified*

We know how the Security Interceptors work, but how do they come to be? How do they know what to intercept? The answer to that lies in the next few components, so keep reading.

The XML Namespace

The XML namespace is of extreme importance to the general appeal and usability of the framework, yet it is, in theory, not strictly necessary. If you know how the Spring Framework's namespaces work, you probably have a good idea of what is going on when you define your security-specific XML configuration in your application context definition files. If you don't know how they work, maybe you think Spring is somehow made aware of how to treat these specific elements and how to load them in the general Spring application context. Either way, here I will explain in some detail the process behind the definition of a custom namespace in Spring, and particularly, the elements in the Spring Security namespace.

Originally, Spring did not support custom XML. All that Spring understood was its own classes defined in the standard Spring Core namespace, where you can define `<bean>`s on a bean-to-bean basis and can't really define anything conceptually more complex without adding that complexity yourself to the configuration.

This `<bean>`–based configuration was, and still is, very good for configuring general-purpose bean instances, but it could get messy really fast for defining more domain-specific utilities. And beyond being messy, it is also very poor at expressing the business domain of the beans you are defining.

I'll explore this manual configuration later in the book, but for standard cases it is not needed, and you should simply use the namespace. However, keep in mind that under the hood the namespace is nothing more than syntactic sugar. At the end of the day, you still end up with standard Spring beans and objects.

Spring 2.0 introduced support for defining custom XML namespaces. Since then, a lot of projects have made use of this facility, making them more attractive to work with.

An XML custom namespace is simply an XML-based Domain Specific Language (DSL), guided by the rules of an XML schema (xsd) file, that allow developers to create Spring beans using concepts and a syntax more in synch with the programming concerns they are trying to model.

■ **Note** A DSL is a language customized to represent the concepts of a particular application domain. Sometimes, a whole new language is created to support the new domain. These are referred to as *external DSLs*. Some other times, an existing language is tweaked to allow for new expressions that represent the concepts of the domain. These are referred to as *internal DSLs*. In the case presented in this chapter, you are using a general-purpose language (XML); however, you are defining certain constraints about the elements (using XSD) and thus are creating an internal DSL to represent security concepts.

To make Spring aware of a new namespace is really simple. (That's not to say it is simple to actually parse the information of the XML and convert it to beans—this depends on the complexity of your DSL.) All you need is the following:

- An xsd file defining your particular XML structure
- A `spring.schemas` file where you specify the mapping between a URL-based schema location and the location of your xsd file in your classpath
- A `spring.handlers` file where you specify which class is in charge of handling everything related to your namespace
- A bunch of parser classes that will be in charge of parsing each of the top elements defined in your XML file

In Chapter 8 you will see some examples of how to create a new namespace element and integrate it with Spring Security.

For Spring Security, all the namespace configuration-related information resides in the `config` module. In Figure 3-6, you can see the expanded structure of the `config` module as seen in the Eclipse integrated development environment (IDE).

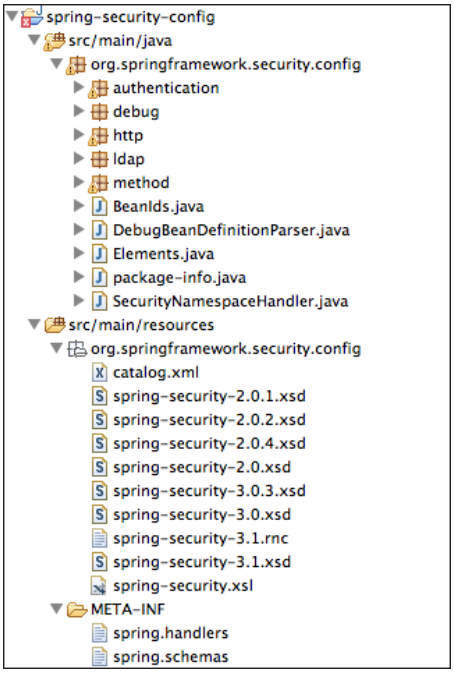


Figure 3-6. *Spring Security's `config` module file structure*

The files `spring.handlers` and `spring.schemas` should reside on the `META-INF` directory in the classpath so that Spring can find them there.

OK, so enough of the general namespace information. More specifically, how does the Spring Security namespace work?

When you create a Spring -based application using XML-defined application context configuration with some of the Spring Security namespace definitions, and you run the application, when it starts to load up, it looks in the application context's namespace definitions at the top of the XML configuration file. It will find the reference to the Spring Security namespace (normally a reference like this `xmlns:security="http://www.springframework.org/schema/security"`). Using the information from the mapping file `spring.handlers`, it will see that the file to handle the security elements is the final class, `org.springframework.security.config.SecurityNamespaceHandler`. Spring calls the parse method of this class for every top element in the configuration file that uses the security namespace. Figure 3-7 shows the load-up sequence for this process.

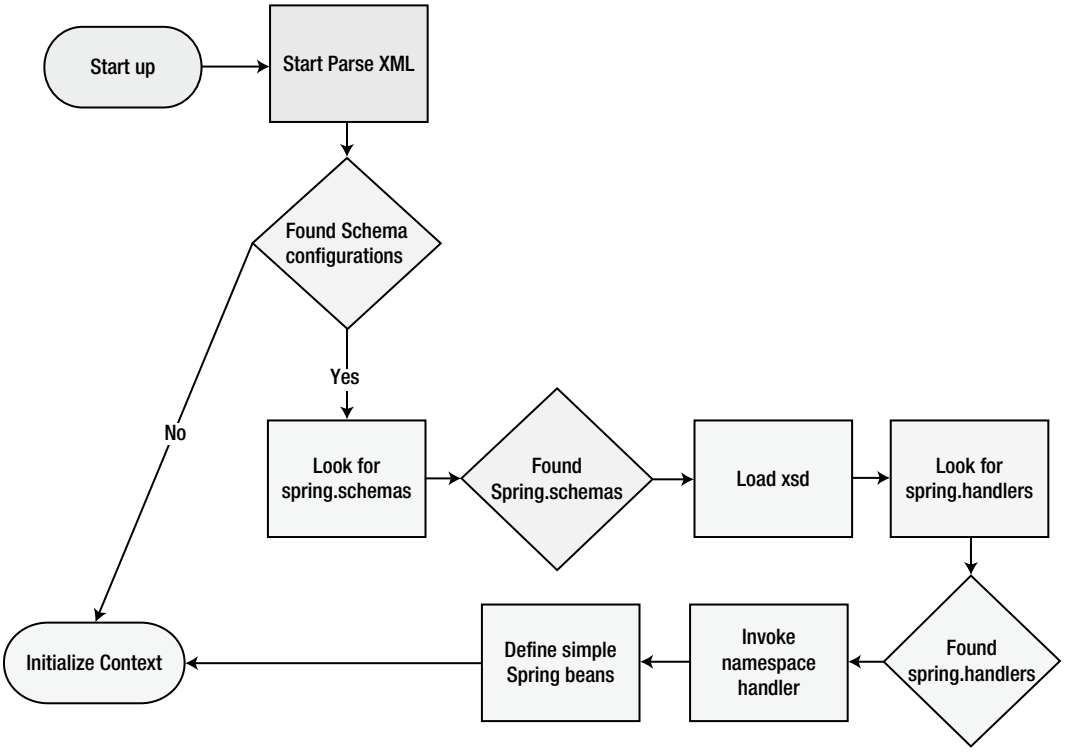


Figure 3-7. *Sequence of loading up a Spring namespace*

`SecurityNamespaceHandler` delegates to a series of `BeanDefinitionParser` objects for the individual parsing of each top-level element. The whole list of elements supported in the Spring Security namespace configuration are defined in the class `org.springframework.security.config.Elements` as constants. This class is shown in Listing 3-2.

Listing 3-2. Constants for all the Spring Security namespace elements

```
package org.springframework.security.config;
public abstract class Elements {

public static final String ACCESS_DENIED_HANDLER = "access-denied-handler";
public static final String AUTHENTICATION_MANAGER = "authentication-manager";
public static final String AFTER_INVOCATION_PROVIDER = "after-invocation-provider";
public static final String USER_SERVICE = "user-service";
public static final String JDBC_USER_SERVICE = "jdbc-user-service";
public static final String FILTER_CHAIN_MAP = "filter-chain-map";
public static final String INTERCEPT_METHODS = "intercept-methods";
public static final String INTERCEPT_URL = "intercept-url";
public static final String AUTHENTICATION_PROVIDER = "authentication-provider";
public static final String HTTP = "http";
public static final String LDAP_PROVIDER = "ldap-authentication-provider";
public static final String LDAP_SERVER = "ldap-server";
public static final String LDAP_USER_SERVICE = "ldap-user-service";
public static final String PROTECT_POINTCUT = "protect-pointcut";
```

Download from Wow! eBook <www.wowebook.com>

```
public static final String EXPRESSION_HANDLER = "expression-handler";
public static final String INVOCATION_HANDLING = "pre-post-annotation-handling";
public static final String INVOCATION_ATTRIBUTE_FACTORY = "invocation-attribute-factory";
public static final String PRE_INVOCATION_ADVICE = "pre-invocation-advice";
public static final String POST_INVOCATION_ADVICE = "post-invocation-advice";
public static final String PROTECT = "protect";
public static final String SESSION_MANAGEMENT = "session-management";
public static final String CONCURRENT_SESSIONS = "concurrency-control";
public static final String LOGOUT = "logout";
public static final String FORM_LOGIN = "form-login";
public static final String OPENID_LOGIN = "openid-login";
public static final String OPENID_ATTRIBUTE_EXCHANGE = "attribute-exchange";
public static final String OPENID_ATTRIBUTE = "openid-attribute";
public static final String BASIC_AUTH = "http-basic";
public static final String REMEMBER_ME = "remember-me";
public static final String ANONYMOUS = "anonymous";
public static final String FILTER_CHAIN = "filter-chain";
public static final String GLOBAL_METHOD_SECURITY = "global-method-security";
public static final String PASSWORD_ENCODER = "password-encoder";
public static final String SALT_SOURCE = "salt-source";
public static final String PORT_MAPPINGS = "port-mappings";
public static final String PORT_MAPPING = "port-mapping";
public static final String CUSTOM_FILTER = "custom-filter";
public static final String REQUEST_CACHE = "request-cache";
public static final String X509 = "x509";
public static final String JEE = "jee";
public static final String FILTER_SECURITY_METADATA_SOURCE = "filter-security-metadata-source";
public static final String METHOD_SECURITY_METADATA_SOURCE = "method-security-metadata-source";
    @Deprecated
public static final String FILTER_INVOCATION_DEFINITION_SOURCE = "filter-invocation-definition-source";
public static final String LDAP_PASSWORD_COMPARE = "password-compare";
public static final String DEBUG = "debug";
public static final String HTTP_FIREWALL = "http-firewall";
}
```

From the list of elements presented in the previous class, the top-level ones as used in the XML configuration files are as follows (in the previous listing, I refer to them by the name of the constant and not by the XML element name):

- `LDAP_PROVIDER`. This element is used to configure the Lightweight Directory Access Protocol (LDAP) authentication provider for your application in case you require one.
- `LDAP_SERVER`. This element is used to configure an LDAP server in your application.
- `LDAP_USER_SERVICE`. This element configures the service for retrieving user details from an LDAP server and populating that user's authorities (Spring Security uses the term "authorities" to refer to the permission names that are granted to a particular user. For example `ROLE_USER` is an authority).
- `USER_SERVICE`. This element defines the in-memory user service where you can store user names, credentials, and authorities directly in the application context definition file.
- `JDBC_USER_SERVICE`. This element allows you to set up a database-driven user service, where you specify a `DataSource` and the queries to retrieve the user information from a database.

- AUTHENTICATION_PROVIDER. This element defines a DaoAuthenticationProvider, which is an authentication provider that delegates to an instance of UserDetailsService. The UserDetailsService can be any of the ones defined in the previous three bullet points, or a reference to a customized one.
- GLOBAL_METHOD_SECURITY. This element is in charge of setting up the global support in your application to the annotations @Secured, @javax.annotation.security.RolesAllowed, @PreAuthorize, and @PostAuthorize. This element will be the one that will handle the registration of a method interceptor that will be aware of all the bean's methods metadata in order to apply the corresponding security advice.
- AUTHENTICATION_MANAGER. This element registers a global ProviderManager in the application and sets up the configured AuthenticationProviders on it.
- METHOD_SECURITY_METADATA_SOURCE. This element registers a MapBasedMethodSecurityMetadataSource in the application context. It will hold a Map<RegisteredMethod, List<ConfigAttribute>>. It does this so that when a request is made to a method, the method can be retrieved and its security constraints can be checked.
- DEBUG. For development purposes, this element registers a DebugFilter in the security filter chain.
- HTTP. This is the main element for a web-based secure application. The HTTP element is really powerful. It allows the definition of URL-based security-mapping strategies, the configuration of the filters, the Secure Sockets Layer (SSL) support and other HTTP-related security configurations.
- HTTP_FIREWALL. This element uses a firewall element and adds it to the filter chain if it is configured. The firewall referenced should be an implementation of Spring's own HttpFirewall interface.
- FILTER_INVOCATION_DEFINITION_SOURCE. This element has been deprecated. See the following one.
- FILTER_SECURITY_METADATA_SOURCE. This element wraps a list of <intercept-url> elements. These elements map the relationship between URLs and the ConfigAttributes required for accessing those URLs.
- FILTER_CHAIN. This element allows you to configure the Spring Security filter chain that will be used in the application, which filters you want to add to the chain, and a request matcher if you want to customize how the chain matches requests. The most important request matchers are: ant based and regexp based.

You will be using the Spring Security namespace thoroughly throughout the book, so many of the elements described here will be revisited in later chapters.

The Filters and Filter Chain

The filter chain model is what Spring Security uses to secure web applications. This model is built on top of the standard *servlet filter* functionality. Working as an Intercepting Filter Pattern, the filter chain in Spring Security is built of a few single-responsibility filters that cover all the different security constraints required by the application.

The filter chain in Spring Security preprocesses and postprocesses all the HTTP requests that are sent to the application and then applies security to URLs that require it.

The Spring Security filter chain is made up of Spring beans; however, standard servlet-based web applications don't know about Spring beans. For this reason, a special servlet filter is needed that can cross the boundaries between the standard servlet API and life cycle and the Spring application where the bean filters will reside. This is the job of the `org.springframework.web.filter.DelegatingFilterProxy` defined in the `web.xml`, which will use under the hood the `WebApplicationContextUtils.getWebApplicationContext` utility method to retrieve the root application context of the application. These two classes are from the Spring Framework, not from Spring Security.

Figure 3-8 shows the configuration of the filter chain.

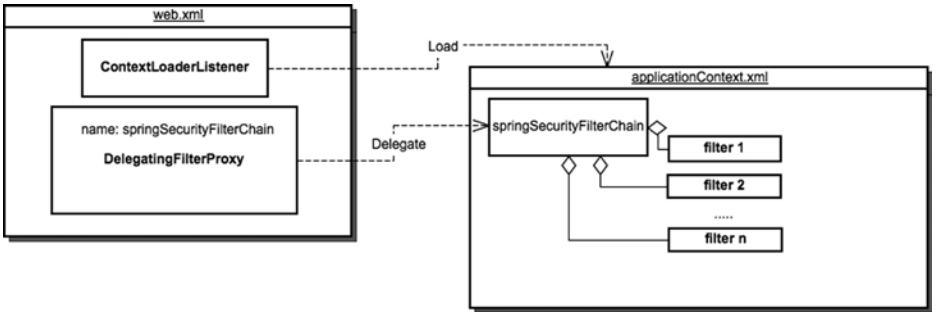


Figure 3-8. *Understanding the Spring Security filter configuration. The filter in the web.xml file has the same name as the bean in the Spring application context so that the listener can find it*

The filter chain will be fully explained in Chapter 4. However, here I'll provide an overview of which filters are available and what they do. The available filters are defined as enums in the file `org.springframework.security.config.http.SecurityFilters`. The enums are then referenced later in the startup process when instantiating the bean definitions for each filter. Here are the defined filters:

- `CHANNEL_FILTER`. This filter ensures that the request is handled by the correct channel—meaning, in most cases, it determines whether or not the request is handled by HTTPS.
- `CONCURRENT_SESSION_FILTER`. This filter is part of the concurrent session-handling mechanism. Its main function is to query the session to see if it has expired (which happens mainly when the maximum number of concurrent sessions per user are reached) and to log out the user if that is the case.
- `SECURITY_CONTEXT_FILTER`. This filter populates `SecurityContextHolder` with a new or existing security context to be used by the rest of the framework.
- `LOGOUT_FILTER`. This filter is based, by default, on a particular URL invocation (`/j_spring_security_logout`). It takes care of the logout process, including tasks such as clearing the cookies, removing the remember-me information, and clearing the security context.
- `X509_FILTER`. This filter extracts the principal and credentials from an X509 certificate using the class `java.security.cert.X509Certificate` and attempts to authenticate with these pre-authenticated values.
- `PRE_AUTH_FILTER`. This filter is used with the J2EE authentication mechanism. The J2EE authenticated principal will be used as the pre-authenticated principal in the framework.
- `FORM_LOGIN_FILTER`. This filter is used when a user name and password is required on a login form. This filter takes care of authenticating with the requested user name and password. It handles requests to a particular URL (`/j_spring_security_check`) with a particular set of user-name and password parameters (`j_username`, `j_password`).

- `OPENID_FILTER`. This filter processes `OpenId` authentication requests, handling both the initial request with the `OpenId` identity to the external server and the redirect from the `OpenId` server back to the application. All this interaction is managed when the filter detects requests to the preconfigured URL `/j_spring_openid_security_check`.
- `LOGIN_PAGE_FILTER`. This filter generates a default login page when the user doesn't provide a custom one. It will be activated when the URL `/spring_security_login` is requested.
- `DIGEST_AUTH_FILTER`. This filter processes HTTP `Digest` authentication headers. It will look for the presence of both `Digest` and `Authorization` HTTP request headers. It can be used to provide Digest authentication to standard user agents, like browsers, or to application clients like SOAP. On successful authentication, the `SecurityContext` will be populated with the valid `Authentication` object.
- `BASIC_AUTH_FILTER`. This `filter` processes the BASIC authentication headers in an HTTP request. It looks for the header `Authorization` and tries to authenticate with these credentials.
- `REQUEST_CACHE_FILTER`. This filter retrieves a request from the request-cache that matches the current request, and it sends the cached one through the rest of the filter chain.
- `SERVLET_API_SUPPORT_FILTER`. This filter wraps the request in a request wrapper that implements the Servlet API security methods, like `isUserInRole`, and delegates it to `SecurityContextHolder`. This allows for the convenient use of the request object itself to get the security information. For example, you can use `request.getAuthentication` to retrieve the `Authentication` object.
- `JAAS_API_SUPPORT_FILTER`. This filter tries to obtain and use `javax.security.auth.Subject`, which is a final class, and continue the filter chain execution with this subject.
- `REMEMBER_ME_FILTER`. If no user is logged in, this filter will look to see whether there is any "remember me" functionality active and any "remember me" `Authentication` available. If there is, this filter will try to login automatically and authenticate with this "remember me" information.
- `ANONYMOUS_FILTER`. This filter checks to see whether there is already an `Authentication` in the context. If there is not, it creates a new `Anonymous` one and sets it on the security context.
- `SESSION_MANAGEMENT_FILTER`. This filter passes the `Authentication` object that corresponds to the authenticated user who is logged in to the system to some configured session management processors in order to do session-related handling of the `Authentication`. Mainly, these processors will do some kind of validation and throw `SessionAuthenticationException` if appropriate. Currently, these processors (or strategies) include only one main class in the form of `org.springframework.security.web.authentication.session.ConcurrentSessionControlStrategy`, dealing with both session fixation and concurrent sessions.
- `EXCEPTION_TRANSLATION_FILTER`. This filter handles the translation between Spring Security exceptions (like `AccessDeniedException`) and the corresponding HTTP status code. It also redirects to the application entry point in case the exception is thrown because there is not yet an authenticated user in the system.
- `FILTER_SECURITY_INTERCEPTOR`. This filter handles the authorization mechanism for defined URLs. It delegates to its parent class' (`AbstractSecurityInterceptor`) functionality (which I'll cover later in the chapter) the actual workflow logic of granting or not granting the access to the specific resource.

- `SWITCH_USER_FILTER`. This filter allows a user to impersonate another one by visiting a particular URL (`/j_spring_security_switch_user`, by default). This URL should be secured to allow just certain users access to this functionality. Also, the method `attemptSwitchUser` in the implementing class `SwitchUserFilter` can be overridden to add constraints, so that you can use more finely grained information to decide if certain users are allowed or not allowed to impersonate other users.

ConfigAttribute

The interface `org.springframework.security.access.ConfigAttribute` encapsulates the access information metadata present in a secured resource. For example, for our study purposes, `ROLE_ADMIN` is a `ConfigAttribute`. There are a few implementations of ConfigAttribute, as you can see in Figure 3-9.

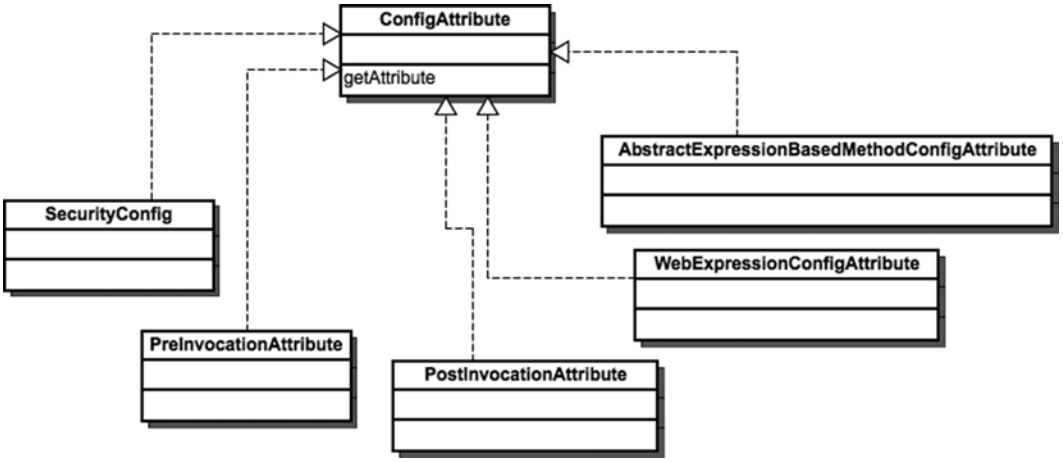


Figure 3-9. *ConfiAttribute hierarchy*

When you annotate a method with `@Secured("ROLE_ADMIN")` or something similar, or specify a URL with `<security:intercept-url pattern="/hello" access="ROLE_SIMPSON_MEMBER" />`, Spring Security does the following. On startup, as normal Spring functionality, all the bean postprocessors in the `ApplicationContext` get invoked. And in the case of Spring Security, the process is the following.

What happens in the case of web requests is not really that complex. Web requests don't really use the postprocessor infrastructure.

When you use the element `<security:intercept-url pattern="/x" access="ROLE_XX" />`, Spring Security uses the class `FilterInvocationSecurityMetadataSourceParser` to parse this XML. In the parsing process, the private method `parseInterceptUrlsForFilterInvocationRequestMap` will be invoked. This method maps the information contained in each of the URL patterns in the XML element into a map of Ant-style request paths, like `/*, ROLE_USER`. Here `/*` is an Ant pattern and `ROLE_USER` is a config attribute (this says basically this config attribute is needed to access any URL with this pattern). This map, ultimately, will be set up in an instance of an implementation of the interface `org.springframework.security.web.access.intercept.FilterInvocationSecurityMetadataSource` inside the `FilterSecurityInterceptor`, which uses it when each request comes to match the requested URL against the keys in the map to find out if the URL is secured and then extracts the `ConfigAttributes` against which to check the authorities of the requesting `Authentication` object.

This setup process is shown in Figure 3-10.

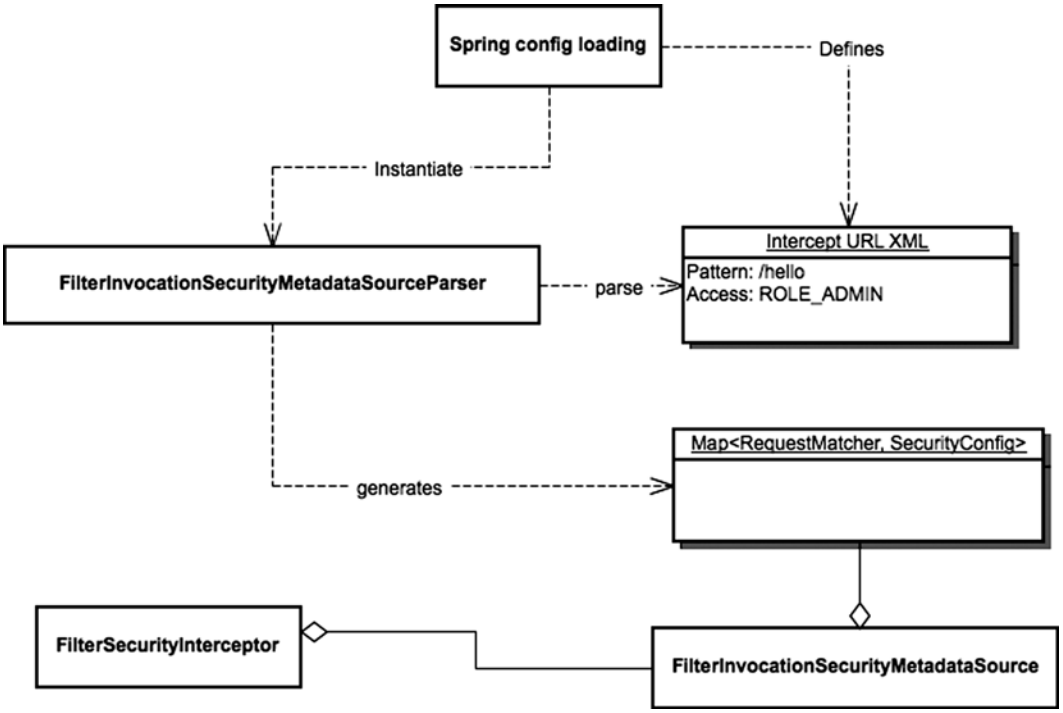


Figure 3-10. *`ConfigAttribute` setup for web applications*

For method-level security, you have many options—the most common one being the configurations performed through the use of annotations. There are a few different annotations available in the framework; however, the setup treatment by the framework is very similar. In the case of the `@Secured` annotation, for instance, you need to make Spring aware that this special annotation needs a special security treatment. To do that, you register the following in the security application context XML file:

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

When you set up that definition in the application context XML configuration, Spring Security creates a new `MethodSecurityMetadataSourceAdvisor` and registers it in the application context. This `advisor will be marked as an infrastructure advisor` and will be picked up by Spring Core's `InfrastructureAdvisorAutoProxyCreator`, which is a `BeanPostProcessor` that Spring initializes automatically. This processes all the beans in the application context and determines if any of the configured advisors can be applied to any of the beans and their methods. If so, it wraps the bean with the required advisor or advisors. The postprocessor finds the `MethodSecurityMetadataSourceAdvisor` and eventually calls an implementation of the `MethodSecurityMetadataSource.getAttributes` method for each bean and all their methods, to determine if they have any `ConfigAttribute` configured as metadata in them. If the `MethodSecurityMetadataSource` finds `ConfigAttributes` in the bean, the `InfrastructureAdvisorAutoProxyCreator`, from Spring Core, calls its own method (`createProxy`) to apply `MethodSecurityMetadataSourceAdvisor`, which internally contains the Security Interceptor as the `org.aopalliance.aop.Advice` to apply to the bean.

Figure 3-11 shows this interaction graphically.

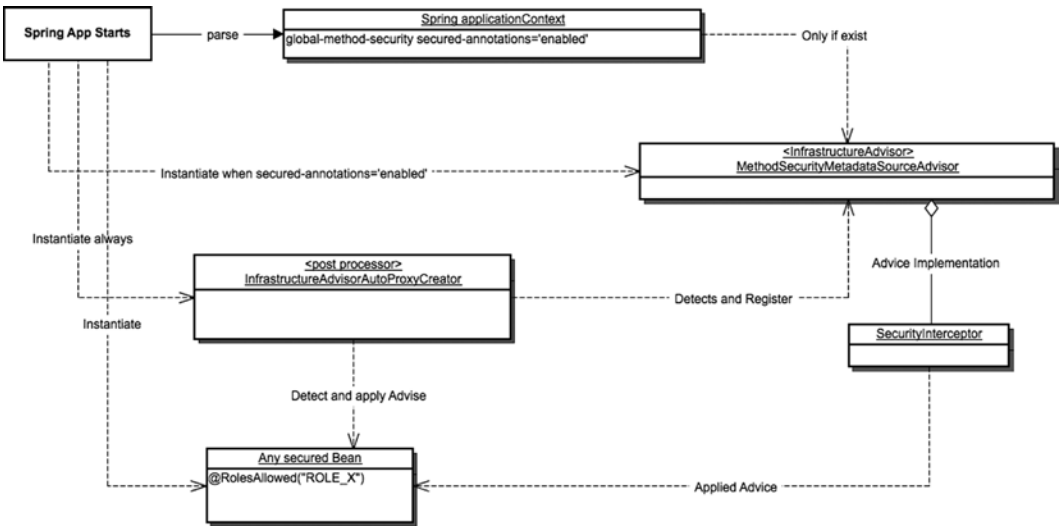


Figure 3-11. *Decorating annotated beans with security Advice*

Again, this is a lot of work that Spring and Spring Security do on your behalf. At first sight, this looks like simple magic, but it takes a lot of hard work from Spring to do it. And you have to thank the Spring and Spring Security developers for taking care of all this and giving you a simple and powerful API for resolving your security concerns.

The Authentication Object

The `Authentication` object is an abstraction that represents the entity that logs in to the system—most likely, a user. Because it is normally a user authenticating, I'll assume and use the term "user" in the rest of the book. There are a few implementations of the `Authentication` object in the framework, as you can see in Figure 3-12.

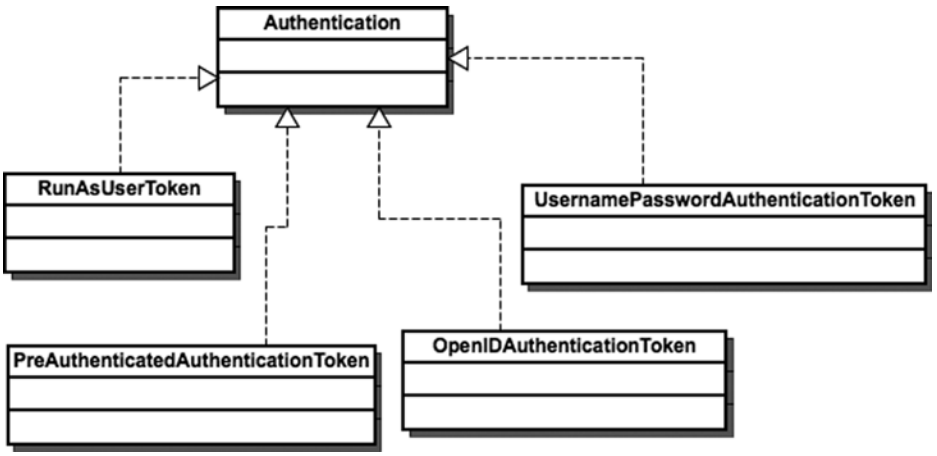


Figure 3-12. *Authentication hierarchy*

An `Authentication` object is used both when an authentication request is created (when a user logs in), to carry around the different layers and classes of the framework the requesting data, and then when it is validated, containing the authenticated entity and storing it in `SecurityContext`.

The most common behavior is that when you log in to the application a new `Authentication` object will be created storing your user name, password, and permissions—most of which are technically known as `Principal`, `Credentials`, and `Authorities`, respectively.

`Authentication` is an interface, and it is pretty simple, as Listing 3-3 shows.

■ **Note** There are many implementations of the `Authentication` interface, and in the book I will be referring most of the time to the general `Authentication` interface when we are not interested in the particular implementation type. Of course when I need to talk about the specifics of an implementation detail I will be referring to the concrete classes.

Listing 3-3. The Authentication interface

```
package org.springframework.security.core;

import java.io.Serializable;
import java.security.Principal;
import java.util.Collection;

import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.core.context.SecurityContextHolder;

public interface Authentication extends Principal, Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();

    Object getCredentials();

    Object getDetails();

    Object getPrincipal();

    boolean isAuthenticated();

    void setAuthenticated(boolean isAuthenticated) throws
            IllegalArgumentException;
}
```

As Figure 3-12 shows, currently there are a few implementations of `Authentication` in the framework:

- `UsernamePasswordAuthenticationToken`. This is a simple implementation that contains, as its name clearly specifies, the user name and password information of the authenticated (or pending authentication) user. It is the most common `Authentication` implementation used throughout the system, as many of the `AuthenticationProvider` objects depend directly on this class.
- `PreAuthenticatedAuthenticationToken`. This implementation exists for handling pre-authenticated `Authentication` objects. Pre-authenticated authentications are those where the actual authentication process is handled by an external system, and Spring Security deals only with extracting the principal (or user) information out of the external system's messages.

- `OpenIDAuthenticationToken`. This is an `Authentication` implementation used specifically for `OpenID` authentication schemes. It is used by both the `OpenID` filter and the `OpenID` authentication provider.
- `RunAsUserToken`. This implementation is used by the `RunAsManager`, which is called by the Security Interceptor, when the accessed resource contains a `ConfigAttribute` that starts with the prefix `'RUN_AS_'`. If there is a `ConfigAttribute` with this value, `RunAsManager` adds new `GrantedAuthorities` to the authenticated user corresponding to the `RUN_AS` value.

SecurityContext and SecurityContextHolder

The interface `org.springframework.security.core.context.SecurityContext` (actually, its implementation `is SecurityContextImpl`) is the place where Spring Security stores the valid `Authentication` object, associating it with the current thread. The `org.springframework.security.core.context.SecurityContextHolder` is the class used to access `SecurityContext` from many parts of the framework. It is built mainly of static methods to store and access `SecurityContext`, delegating to configurable strategies the way to handle this `SecurityContext`—for example, one `SecurityContext` per thread (default), one global `SecurityContext`, or a custom strategy. The class diagram for these classes can be seen in Figure 3-13, and Listings 3-4 and 3-5 show the two classes.

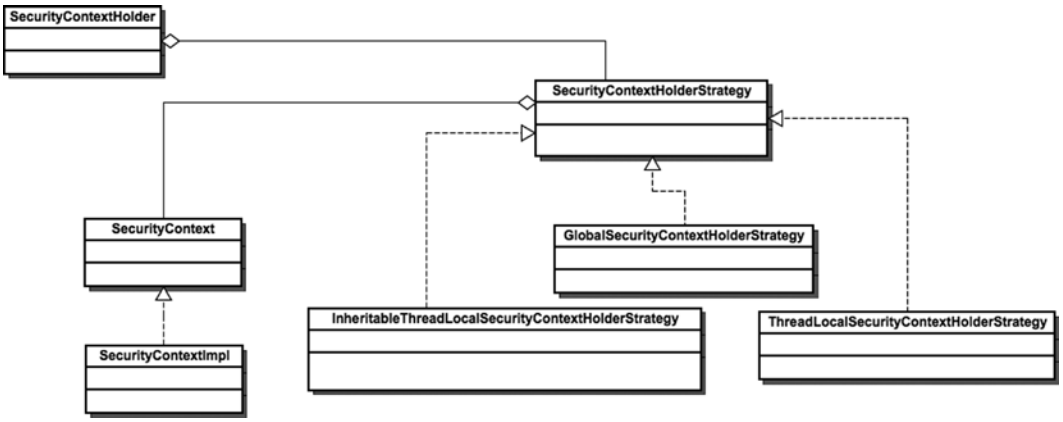


Figure 3-13. *SecurityContext and SecurityContextHolder*

Listing 3-4. SecurityContext interface

```
package org.springframework.security.core.context;
import org.springframework.security.core.Authentication;
import java.io.Serializable;

public interface SecurityContext extends Serializable {
    Authentication getAuthentication();
void setAuthentication(Authentication authentication);
}
```

Listing 3-5. SecurityContextHolder class

```
package org.springframework.security.core.context;

import org.springframework.util.ReflectionUtils;
```

```
import java.lang.reflect.Constructor;

public class SecurityContextHolder {
    public static final String MODE_THREADLOCAL = "MODE_THREADLOCAL";
    public static final String MODE_INHERITABLETHREADLOCAL =
            "MODE_INHERITABLETHREADLOCAL";
    public static final String MODE_GLOBAL = "MODE_GLOBAL";
    public static final String SYSTEM_PROPERTY = "spring.security.strategy";
    private static String strategyName = System.getProperty(SYSTEM_PROPERTY);
    private static SecurityContextHolderStrategy strategy;
    private static int initializeCount = 0;

    static {
        initialize();
    }

    public static void clearContext() {
        strategy.clearContext();
    }

    public static SecurityContext getContext() {
        return strategy.getContext();
    }

    public static int getInitializeCount() {
        return initializeCount;
    }

    private static void initialize() {
        if ((strategyName == null) || "".equals(strategyName)) {
             strategyName = MODE_THREADLOCAL;
        }
        if (strategyName.equals(MODE_THREADLOCAL)) {
            strategy = new ThreadLocalSecurityContextHolderStrategy();
        } else if (strategyName.equals(MODE_INHERITABLETHREADLOCAL)) {
            strategy = new
                    InheritableThreadLocalSecurityContextHolderStrategy();
        } else if (strategyName.equals(MODE_GLOBAL)) {
            strategy = new GlobalSecurityContextHolderStrategy();
        } else {
            try {
                Class<?> clazz = Class.forName(strategyName);
                Constructor<?> customStrategy = clazz.getConstructor();
                strategy = (SecurityContextHolderStrategy) customStrategy
                        .newInstance();
            } catch (Exception ex) {
                ReflectionUtils.handleReflectionException(ex);
            }
        }
        initializeCount++;
    }
```

Download from Wow! eBook <www.wowebook.com>

```
    public static void setContext(SecurityContext context) {
        strategy.setContext(context);
    }

    public static void setStrategyName(String strategyName) {
        SecurityContextHolder.strategyName = strategyName;
        initialize();
    }

    public static SecurityContextHolderStrategy getContextHolderStrategy() {
        return strategy;
    }

    public static SecurityContext createEmptyContext() {
        return strategy.createEmptyContext();
    }

    public String toString() {
        return "SecurityContextHolder[strategy='" + strategyName +
                "';initializeCount = " + initializeCount + "]";
    }
}
```

AuthenticationProvider

`AuthenticationProvider` is the main entry point for authenticating an `Authentication` object. This interface has only two methods, as Listing 3-6 shows. This is one of the major extension points of the framework, as you can tell by the many classes that currently extend this interface. Each of the implementing classes deals with a particular external provider to authenticate against. So if you come across a particular provider that is not supported and need to authenticate against it, you probably need to implement this interface with the required functionality. You will see examples of this later in the book.

Here are some of the existing providers that come with the framework:

```
CasAuthenticationProvider
JaasAuthenticationProvider
DaoAuthenticationProvider
OpenIDAuthenticationProvider
RememberMeAuthenticationProvider
LdapAuthenticationProvider
```

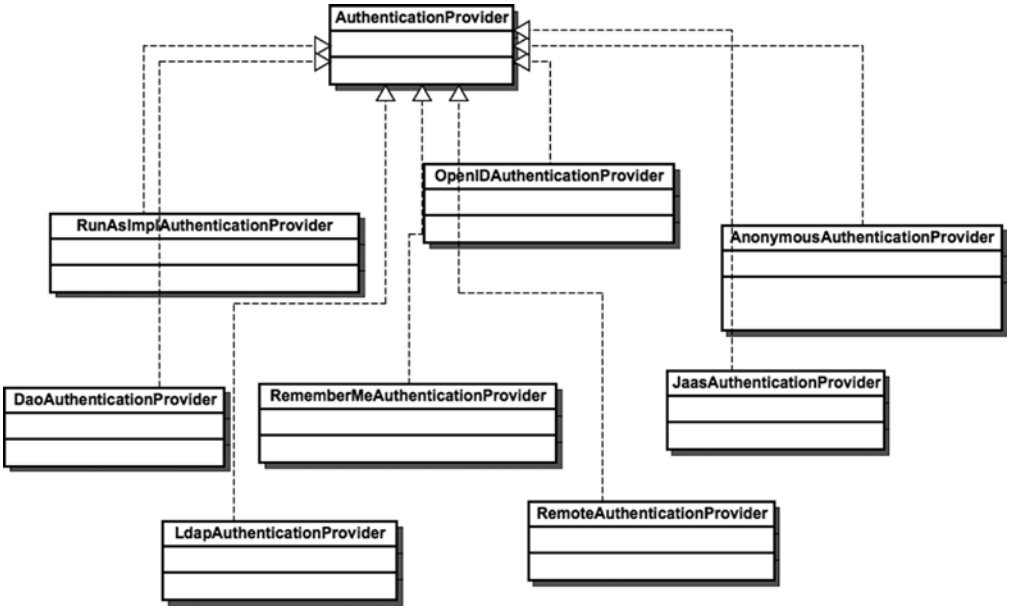


Figure 3-14. *AuthenticationProvider hierarchy*

Listing 3-6. AuthenticationProvider interface

```
package org.springframework.security.authentication;

import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;

public interface AuthenticationProvider {
    Authentication authenticate(Authentication authentication) throws
            AuthenticationException;

    boolean supports(Class<?> authentication);
}
```

AccessDecisionManager

`AccessDecisionManager` is the class in charge of deciding if a particular `Authentication` object is allowed or not allowed to access a particular resource. In its main implementations, it delegates to `AccessDecisionVoter` objects, which basically compares the `GrantedAuthorities` in the `Authentication` object against the `ConfigAttribute`(s) required by the resource that is being accessed, deciding whether or not access should be granted. They emit their vote to allow access or not. The `AccessDecisionManager` implementations take the output from the voters into consideration and apply a determined strategy on whether or not to grant access. Voters, however, also can abstain from voting.

The `AccessDecisionManager` interface can be seen in Listing 3-7. Its UML class diagram is shown in Figure 3-15.

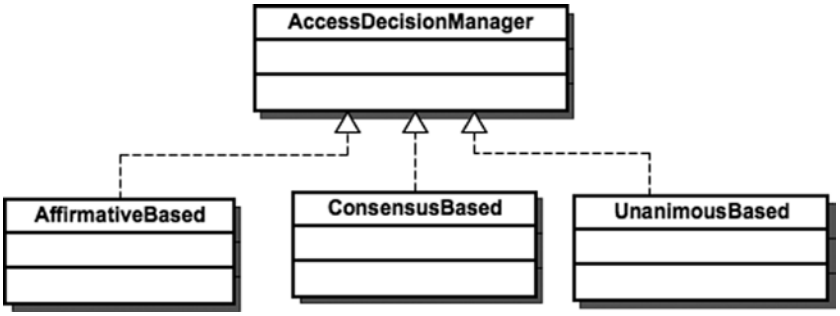


Figure 3-15. *AccessDecisionManager hierarchy*

Listing 3-7. AccessDecisionManager

```
package org.springframework.security.access;

import java.util.Collection;

import org.springframework.security.authentication
        .InsufficientAuthenticationException;
import org.springframework.security.core.Authentication;

public interface AccessDecisionManager {
    void decide(Authentication authentication, Object object,
                Collection<ConfigAttribute> configAttributes) throws
            AccessDeniedException, InsufficientAuthenticationException;

    boolean supports(ConfigAttribute attribute);

    boolean supports(Class<?> clazz);
}
```

The current `AccessDecisionManager` implementations all delegate to voters, but they work in slightly different ways. The current voters, which are described in the following list, are defined in the package `org.springframework.security.access.vote`.

AffirmativeBased

This access decision manager calls all its configured voters, and if any of them votes that access should be granted, it is enough for the access decision manager to allow access to the secured resource. If no voters vote to grant access and there is at least one voting not to grant it, the access decision manager throws an `AccessDeniedException` denying access. If there are only abstaining voters, a decision is made based on the `AccessDecisionManager`'s instance variable `allowIfAllAbstainDecisions`, which is a Boolean that defaults to false, determining if access should be granted or not when all voters abstain.

ConsensusBased

This access decision manager implementation calls all its configured voters to make a decision to either grant or deny access to a resource. The difference with the `AffirmativeBased` decision manager is that the `ConsensusBased` decision manager decides to grant access only if there are more voters granting access than voters denying it. So the majority wins in this case. If there are the same number of granting voters as denying voters, the value of the instance variable

allowIfEqualGrantedDeniedDecisions is used to decide. By default, this variable's value is "true", access is granted. When all voters abstain, the access decision will be decided the same way as it is for the AffirmativeBased manager.

UnanimousBased

As you probably guessed, this access decision manager will grant access to the resource only if all the configured voters vote in favor of allowing access to the resource. If any voter votes to deny the access, the AccessDeniedException will be thrown. The "all abstain" case is handled the same way as with the other implementations of AccessDecisionManager.

AccessDecisionVoter

This discussion of the AccessDecisionManager and its current implementations should have made clear the importance of the Access Decision Voters, because they are the ones, working as a team, that ultimately determine if a particular Authentication object has enough privileges to access a particular resource.

The org.springframework.security.access.AccessDecisionVoter interface is very simple as well, and you can see it in Listing 3-8.

The main method is "vote", and as can be deduced from the interface, it will return one of three possible responses (ACCESS_GRANTED, ACCES_ABSTAIN, ACCESS_DENIED), depending on whether the required conditions are satisfied.

The satisfaction or not of the conditions is given by analyzing the Authentication object's rights against the required resource. In practice this basically means that the Authentication's authorities are compared against the resource's security attributes looking for matches.

Following are the current AccessDecisionVoter implementations:

- org.springframework.security.access.annotation.Jsr250Voter. This voter votes on resources that are secured with JSR 250 annotations—namely, DenyAll, PermitAll, and RolesAllowed. Their names should be very descriptive. DenyAll won't allow any access at all to the resource, independent of the security information carried by the Authentication object trying to access it. PermitAll will allow access to everyone, regardless of what roles they have. The RolesAllowed annotation can be configured with a series of roles. If an Authentication object tries to access the resource, it must have one of the roles configured in the RolesAllowed annotation in order to get access granted by this voter.

- org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter. This voter votes on resources with expression configurations based on @PreFilter and @PreAuthorize annotations. @PreFilter and @PreAuthorize annotations support a value attribute that can have a SpEL expression. The PreInvocationAuthorizationAdviceVoter is the one in charge of evaluating the SpEL expressions (of course with the help of Spring's SpEL evaluation mechanism) provided in these annotations. We will be explaining and using SpEL expressions in several parts of the book so this concept will become clearer as the books advances.

- org.springframework.security.access.vote.AbstractAclVoter. This is the abstract class that has the skeleton to write voters dealing with domain ACL rules so that other implementing class built on its functionality to add voting behavior. Currently, it is implemented in AclEntryVoter, which votes on users' permissions on domain objects. This voter will be covered in the chapter dedicated to ACL.

- org.springframework.security.access.vote.AuthenticatedVoter. This voter votes whenever a ConfigAttribute referencing any of the three possible levels of authentication is present on the secured resource. The three levels are IS_AUTHENTICATED_FULLY, IS_AUTHENTICATED_REMEMBERED, IS_AUTHENTICATED_ANONYMOUSLY. The voter emits a positive vote if the Authentication object's authentication level matches (or is a stronger level in the hierarchy IS_AUTHENTICATED_FULLY ➤ IS_AUTHENTICATED_REMEMBERED ➤ IS_AUTHENTICATED_ANONYMOUSLY) the authentication level configured in the resource.

- `org.springframework.security.access.vote.RoleVoter`. This is, perhaps, the most commonly used voter of them all. This voter, by default, is able to vote on resources that have `ConfigAttribute`(s) containing security metadata starting with the prefix "ROLE_" (which can be overridden). When an `Authentication` object tries to access the resource, its `GrantedAuthorities` will be matched against the relevant `ConfigAttributes`. If there is a match, access is granted. If there isn't, access is denied.
- `org.springframework.security.access.expression.WebExpressionVoter`. This is the voter in charge of evaluating SpEL expressions in the context of web requests in the filter chain—expressions like `'hasRole'` in the `<intercept-url>` element. To make use of this voter, and in general to support SpEL expressions in web security, `the use-expressions="true"` attribute needs to be added to the `<http>` element.

The voters model is yet another one in the framework that is open for extension and customization. You could easily create your own implementation and add it to the framework. You will see how to do this in Chapter 8.

Listing. 3-8. AccessDecisionVoter interface

```
package org.springframework.security.access;

import java.util.Collection;

import org.springframework.security.core.Authentication;

public interface AccessDecisionVoter<S> {
    int ACCESS_GRANTED = 1;
    int ACCESS_ABSTAIN = 0;
    int ACCESS_DENIED = -1;

    boolean supports(ConfigAttribute attribute);

    boolean supports(Class<?> clazz);

    int vote(Authentication authentication, S object,
             Collection<ConfigAttribute> attributes);
    }
```

UserDetailsService and AuthenticationUserDetailsService

The interface `org.springframework.security.core.userdetails.UserDetailsService` is in charge of loading the user information from the underlying user store (in-memory, database, and so on) when an authentication request arrives in the application. `UserDetailsService` makes use of the provided `user name` for looking up the rest of the required user data from the datastore. It defines just one method, as you see in Listing 3-9. You can see its hierarchy in Figure 3-16.

Listing 3-9. UserDetailsServicepackage org.springframework.security.core.userdetails

```
public interface UserDetailsService {
  UserDetails loadUserByUsername(String username) throws   UsernameNotFoundException;
}
```

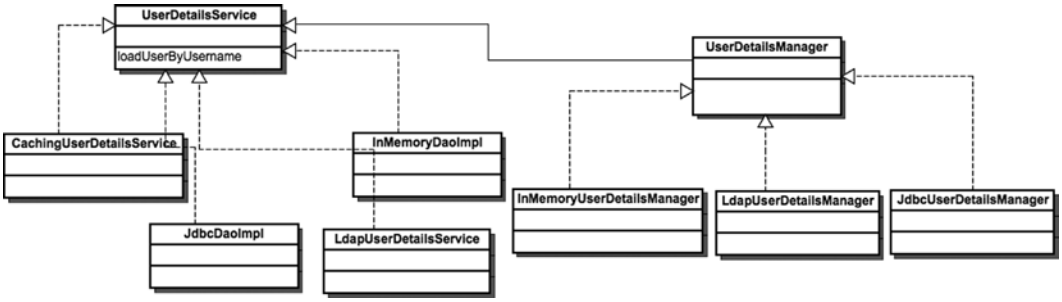


Figure 3-16. `UserDetailsService` *hierarchy*

The `interface org.springframework.security.core.userdetails.AuthenticationUserDetailsService` is more generic—it allows you to retrieve a `UserDetails` using an `Authentication` object instead of a `user name String`, making it more flexible to implement. Actually, there is an implementation of `AuthenticationUserDetailsService` (`UserDetailsByNameServiceWrapper`) that simply delegates to a `UserDetailsService` extracting the `user name` from the `Authentication` object.

Listing 3-10 shows the `AuthenticationUserDetailsService` interface.

These are the two main strategies (`AuthenticationUserDetailsService and UserDetailsService`) used for retrieving the user information when attempting authentication. They are usually called from the particular `AuthenticationProvider` that is being used in the application. For example, the `OpenIDAuthenticationProvider` and `CasAuthenticationProvider` delegate to an `AuthenticationUserDetailsService` to obtain the user details, while the `DaoAuthenticationProvider` delegates directly to a `UserDetailsService`. Some other providers don't use a user details service of any kind (for example, `JaasAuthenticationProvider` uses its own mechanism to retrieve the Principal from a `javax.security.auth.login.LoginContext`), and some others use a completely custom one (for example, `LdapAuthenticationProvider` uses a `UserDetailsContextMapper`).

Listing 3-10. AuthenticationUserDetailsService

```
package org.springframework.security.core.userdetails;

public interface AuthenticationUserDetailsService<T extends Authentication> {
    UserDetails loadUserDetails(T token) throws UsernameNotFoundException;
}
```

UserDetails

The interface `org.springframework.security.core.userdetails.UserDetails` object is the main abstraction in the system, and it's used to represent a full user in the context of Spring Security. It is also made available to be accessed later in the system from any point that has access to `SecurityContext`. Normally, developers create their own implementation of this interface to store particular user details they need or want (like email, telephone, address, and so on). Later, they can access this information, which will be encapsulated in the `Authentication object`, and they can be obtained by calling the `getPrincipal` method on it.

Some of the current `UserDetailsService` (for example, `InMemoryUserDetailsManager`) implementations use the class `org.springframework.security.core.userdetails.User`, `which is` available in the core of the framework, as the `UserDetails` implementation returned by the method `loadUserByUsername`. However, this is another of those configurable points of the framework, and you could easily create your own `UserDetails implementation` and use that in your application. Listing 3-11 shows the `UserDetails` interface.

Listing 3-11. UserDetails interface

```
package org.springframework.security.core.userdetails;

import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;

import java.io.Serializable;
import java.util.Collection;

public interface UserDetails extends Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();

    String getPassword();

    String getUsername();

    boolean isAccountNonExpired();

    boolean isAccountNonLocked();

    boolean isCredentialsNonExpired();

    boolean isEnabled();
}
```

ACL

The ACL is the module in charge of securing your application at the individual domain object level with a fine level of granularity. This means, in a general way, assigning an ID to each domain object in your application and creating a relationship between these objects and the different users of the application. These relationships determine whether or not a determined user is allowed access to a particular domain. The ACL model offers a fine-grained, access-level configuration you can use to define different rules for accessing the objects depending on who is trying to access it. (For example, a user might be allowed read access while another user will have write/read access over the same domain object.)

The current support for ACLs is configured to get the configuration rules from a relational database. The DDL (Data Definition Language) for configuring the database comes along with the framework itself, and it can be found in the ACL module.

ACL security will be fully covered in Chapter 7.

JSP Taglib

If you are working to secure a Java web application, the `taglib` component of the framework is the one you use to hide or show certain elements in your pages according to your users' permissions.

The tags are simple to use and, at the same time, very convenient for making a more usable web site. They help you adapt the UI of your application on a per-user (or more commonly, per-role) basis.

The `taglib` will be covered in depth in Chapter 4.

Good Design and Patterns in Spring Security

I said it before, but I will repeat it here. One of the great aspects of working with open source software is that you can (and I would say *you should*) look at the source code and understand the software at a new, deeper level. Also, you can look at the way the software is built, at what is good, and at what is bad (at least by your own subjective standards) and just learn how other developers work. This can have a great impact on the way you work, because you might discover a way of doing things that you couldn't have learned on your own.

Sometimes, of course, you will find things you don't like, but that is good as well. You can learn from other people's mistakes as much as you can learn from their successes.

For me, Spring in general and Spring Security in particular have achieved something that I found invaluable in the Java development space—that is, they can make us better developers even without us noticing it. For instance, I often ask myself, "How many people would be using a template pattern for accessing databases if they weren't using Spring, instead of a more awkward DB access layer?" or "How many people would be just programming against implementation classes all the time, creating unnecessary coupling if it wasn't for Spring's DI support?" or "How many people would have cross-cutting concerns, like transactions, all over their code base if it wasn't for how easily Spring brings AOP into the development process?."

I think helping good practices almost without noticing is really a great achievement for Spring. It won't create great developers by itself for sure, but it helps the average developer to not make mistakes that he might make if he didn't have the support of the framework, and its principles to adhere to.

As you might see from the description of the main components of the framework, Spring Security itself is built with good design principles and patterns in mind. You'll have a brief look here at some of the things I find interesting in the framework, and from which you can learn about.

This section won't really help you to do more with Spring Security, but it will serve as a way to appreciate the good work that has been done in constructing this fantastic framework.

Strategy Pattern

A big part of the pluggability and modularity of the framework is achieved thanks to the wide use of the Strategy pattern. You can find it, for example, in the type of `SecurityContext` to be used, the `AuthenticationProvider` hierarchy, the `AccessDecisionVoters`, and many other elements. Covering design patterns is outside the scope of this book but as a reminder of the strategy pattern's power I leave you with this definition from the Wikipedia. "The strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it".

That definition shows a great deal of the power that comes when working with interfaces. You could have different implementations of the same interface and pass any of them to a client class for doing different kinds of work. The client classes don't need to know or care about the implementation details that it is working with. Knowing the interface or contract is enough to leverage its job.

Decorator Pattern

Built into Spring's core AOP framework, you can find the Decorator pattern—mostly in the way that your annotated business classes and methods get security constraints applied to them. Basically, your objects have only certain meta information related to the security constraints that should be applied to them, and then by some "Spring magic" they get wrapped with security handling. Listing 3-12 shows the invoke method of `MethodSecurityInterceptor`. In the listing, you can see how the objects are decorated with prefunctionality and postfunctionality that surrounds the invocation of the actual method.

Listing 3-12. MethodSecurityInterceptor's invoke method

```
public Object invoke(MethodInvocation mi) throws Throwable {
    InterceptorStatusToken token = super.beforeInvocation(mi);
    Object result = mi.proceed();
    returnsuper.afterInvocation(token, result);
}
```

SRP

Spring Security's code seems to take very seriously the Single Responsibility Principle. There are many examples of it around the framework, because any object you choose seems to have one and only one responsibility. For example, the `AuthenticationProvider` deals only with the general concern of authenticating a principal with its credentials in the system. The `SecurityInterceptor` is simply in charge of intercepting the requests, and it delegates all security-checking logic to collaborating objects. A lot more examples like this can be extracted from the framework.

DI

Again, this is built into the Spring Framework itself, and of course as everything in the Spring architecture, this means that it is also inherited by the rest of the Spring projects, including Spring Security. Dependency injection (DI) is one of Spring's most important features. Almost every component in Spring Security is configured through the use of dependency injection. The `AccessDecisionManager` is injected into the `AbstractSecurityInterceptor`, and `AccessDecisionVoter` implementations are injected into the `AccessDecisionManager`. And like this, most of the framework is built by composing components together through dependency injection.

Summary

This was a complex chapter, but going through the inner workings of a software tool is definitely the best way to understand it and take advantage out of it. And that is what we did.

You looked at an in-depth explanation of Spring Security's architecture, its major components, and the way it works from the inside.

You should now understand how the XML namespace works, how AOP fits into the framework, and how, in general, the Servlet Filter functionality is used to enforce web-level security.

I demystified the "Spring magic" by going through all the components that help you add security to your applications in a seemingly simple way.

You looked at some code snippets from the framework itself to get a greater appreciation of the work done in it, as well as to understand better why things work the way they do.

You also studied the modularity inherent in the framework and saw how it helps to create software that is both flexible and extensible.

Even with all we covered in this chapter, it is basically an introduction and a reference to have in hand when you read the upcoming chapters and you start looking at the options to secure your applications. From now on, you will understand where everything fits in the framework and how the different components link to each other.

In the next chapter, you will start developing an example application. At the beginning, it will be a simple web application, and you will see how to secure it. You will use all your current knowledge of the framework to tweak the configuration and test different ways of implementing security at the web level.

CHAPTER 4

Web Security

In this chapter, I will explain how to apply security at the web layer for Java web-based applications. You will see in detail the inner work of the security filter chain and the different metadata options at your disposal to define security constraints in your application. I will also cover the Taglib facility for enforcing security constraints at the view level.

Introducing the Simple Example Application

In this chapter, you will be working on a dummy test application: an "action movies" web application. The application itself will be very simple. (It won't have any really useful functionality). However, I will try to cover most of the options available for securing web applications with Spring Security, even if some of the options don't seem realistic for an application of this kind.

First, we will set up the application. As in Chapter 2, we will use Maven to start up and build the project. Again, I will assume certain versions of the different tools we need. Mainly, we will use Java 7 and Maven 3.

Go to the command line, to a directory you like, and do the following:

```
mvn archetype:generate -DgroupId=com.apress.pss
    -DartifactId=terrormovies -DarchetypeArtifactId=maven-archetype-webapp
```

Press enter on each prompt you get from the command line. You should see output like Figure 4-1.

Download from Wow! eBook <www.wowebook.com>

```
Carlos-MacBook-Air:ch05 cscarioni$ mvn archetype:generate -DgroupId=com.apress.pss -DartifactId=terrormovies -DarchetypeArtifactId=maven-archetype-webapp
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom >>>
[INFO]
[INFO] <<< maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom <<<
[INFO]
[INFO] --- maven-archetype-plugin:2.2:generate (default-cli) @ standalone-pom ---
[INFO] Generating project in Interactive mode
[INFO] Using property: groupId = com.apress.pss
[INFO] Using property: artifactId = terrormovies
Define value for property 'version':  1.0-SNAPSHOT: :
[INFO] Using property: package = com.apress.pss
Confirm properties configuration:
groupId: com.apress.pss
artifactId: terrormovies
version: 1.0-SNAPSHOT
package: com.apress.pss
 Y: :
[INFO] ----------------------------------------------------------------------------
[INFO] Using following parameters for creating project from Old (1.x) Archetype: maven-archetype-webapp:1.0
[INFO] ----------------------------------------------------------------------------
[INFO] Parameter: groupId, Value: com.apress.pss
[INFO] Parameter: packageName, Value: com.apress.pss
[INFO] Parameter: package, Value: com.apress.pss
[INFO] Parameter: artifactId, Value: terrormovies
[INFO] Parameter: basedir, Value: /Users/cscarioni/projects/pro-spring-security/src/ch05
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] project created from Old (1.x) Archetype in dir: /Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 58.238s
[INFO] Finished at: Sun Dec 16 11:45:22 GMT 2012
[INFO] Final Memory: 11M/140M
[INFO] ------------------------------------------------------------------------
```

Figure 4-1. *Creating a new Maven web application*

Now we have an empty web project. At the moment, we will use the same dependencies that we had in our program from Chapter 2. So make sure your pom.xml file has the content from Listing 4-1.

Listing 4-1. First pom.xml for terrormovies

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.apress.pss</groupId>
  <artifactId>terrormovies</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>terrormovies Maven Webapp</name>
  <url>http://maven.apache.org</url>
          <dependencies>
                <dependency>
                        <groupId>javax.servlet</groupId>
                        <artifactId>javax.servlet-api</artifactId>
                        <version>3.0.1</version>
                </dependency>
                <dependency>
```

```
                <groupId>junit</groupId>
                <artifactId>junit</artifactId>
                <version>4.5</version>
                <scope>test</scope>
            </dependency>
            <dependency>
                <groupId>org.springframework.security</groupId>
                <artifactId>spring-security-core</artifactId>
                <version>3.1.3.RELEASE</version>
            </dependency>
            <dependency>
                <groupId>org.springframework.security</groupId>
                <artifactId>spring-security-config</artifactId>
                <version>3.1.3.RELEASE</version>
            </dependency>
            <dependency>
                <groupId>org.springframework.security</groupId>
                <artifactId>spring-security-web</artifactId>
                <version>3.1.3.RELEASE</version>
            </dependency>
            <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-web</artifactId>
                <version>3.0.7.RELEASE</version>
            </dependency>
             <dependency>
                <groupId>org.springframework</groupId>
                <artifactId>spring-webmvc</artifactId>
                <version>3.0.7.RELEASE</version>
            </dependency>
            <dependency>
                <groupId>commons-logging</groupId>
                <artifactId>commons-logging</artifactId>
                <version>1.1.1</version>
            </dependency>
            <dependency>
                <groupId>commons-codec</groupId>
                <artifactId>commons-codec</artifactId>
                <version>1.3</version>
            </dependency>
        </dependencies>
        <build>
            <plugins>
                <plugin>
                    <groupId>org.mortbay.jetty</groupId>
                    <artifactId>jetty-maven-plugin</artifactId>
                    <version>8.1.1.v20120215</version>
                </plugin>
            </plugins>
            <finalName>terrormovies</finalName>
        </build>
</project>
```

Just to test that it works, go to the command line, and execute the following in the root of the application:

```
mvn jetty:run
```

■ **Note** Jetty (http://jetty.codehaus.org/jetty/ and http://www.eclipse.org/jetty/) is both an HTTP server and a Java Servlet container in the same way as Apache Tomcat. It is a powerful, flexible, open source server that can be used either standalone or embedded in applications. I will use Jetty in all the examples of this book because I find it very convenient to work with in Maven environments. With that said, most, if not all, of the examples should work as they are in any other Servlet container, such as Tomcat.

The application should start without a problem, and you should get output like that shown in Figure 4-2. If you open your browser and go to http://localhost:8080/, you will see the message "Hello World!" on the page, as you can see in Figure 4-3.

```
[INFO] ------------------------------------------------------------------------
[INFO] Building terrormovies Maven Webapp 0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] >>> jetty-maven-plugin:8.1.1.v20120215:run (default-cli) @ terrormovies >>>
[INFO]
[INFO] --- maven-resources-plugin:2.4.3:resources (default-resources) @ terrormovies ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:compile (default-compile) @ terrormovies ---
[INFO] No sources to compile
[INFO]
[INFO] --- maven-resources-plugin:2.4.3:testResources (default-testResources) @ terrormovies ---
[WARNING] Using platform encoding (UTF-8 actually) to copy filtered resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory /Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:2.3.2:testCompile (default-testCompile) @ terrormovies ---
[INFO] No sources to compile
[INFO]
[INFO] <<< jetty-maven-plugin:8.1.1.v20120215:run (default-cli) @ terrormovies <<<
[INFO]
[INFO] --- jetty-maven-plugin:8.1.1.v20120215:run (default-cli) @ terrormovies ---
[INFO] Configuring Jetty for project: terrormovies Maven Webapp
[INFO] webAppSourceDirectory /Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/webapp does not exist. Defaulting to /Users/cscarioni/projects/pro-
spring-security/src/ch05/terrormovies/src/main/webapp
[INFO] Reload Mechanic: automatic
[INFO] Classes = /Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/target/classes
[INFO] Context path = /
[INFO] Tmp directory = /Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/target/tmp
[INFO] Web defaults = org/eclipse/jetty/webapp/webdefault.xml
[INFO] Web overrides =  none
[INFO] web.xml file = file:/Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/webapp/WEB-INF/web.xml
[INFO] Webapp directory = /Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/webapp
2012-12-16 11:53:25.574:INFO:oejs.Server:jetty-8.1.1.v20120215
2012-12-16 11:53:26.042:INFO:oejpw.PlusConfiguration:No Transaction manager found - if your webapp requires one, please configure one.
2012-12-16 11:53:27.512:INFO:oejsh.ContextHandler:started o.m.j.p.JettyWebAppContext{/,file:/Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/weba
pp/},file:/Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/webapp/
2012-12-16 11:53:27.513:INFO:oejsh.ContextHandler:started o.m.j.p.JettyWebAppContext{/,file:/Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/weba
pp/},file:/Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/webapp/
2012-12-16 11:53:27.513:INFO:oejsh.ContextHandler:started o.m.j.p.JettyWebAppContext{/,file:/Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/weba
pp/},file:/Users/cscarioni/projects/pro-spring-security/src/ch05/terrormovies/src/main/webapp/
2012-12-16 11:53:27.613:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Figure 4-2. *Starting the Jetty application*

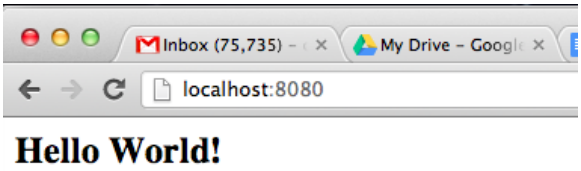


Figure 4-3. *First iteration of the application displays the simple Hello World! page*

Let's start to write some code. In this chapter, you will be working with Spring MVC instead of simple Servlets.

Spring MVC is a simple but powerful Model-View-Controller framework that is part of the Spring Framework. It integrates fully with its programming model. MVC frameworks offer the developer a good separation of concerns, and many web frameworks in different programming languages enforce this programming model.

Covering Spring MVC in depth is outside the scope of this book, but the examples used in the book should be easy enough to follow. If you want to learn about Spring MVC and Spring in general, you should pick up a book like *Pro Spring 3* by Clarence Ho and Rob Harrop (Apress, 2012) or *Spring Recipes: A Problem-Solution Approach, Second Edition* by Gary Mak, Daniel Rubio, and Josh Long (Apress 2010), which cover a lot of ground in the Spring Framework ecosystem.

The first thing you will do is set up Spring MVC in your application. Follow these simple steps:

1. Add the content of Listing 4-2 to your web.xml file. This code snippet defines a Servlet that will be used to handle all incoming requests to your application. You can see that its implementation is a Spring-specific class. This class is the entry point into the Spring MVC world. The `<servlet-name>` value given to the servlet is relevant because `DispatcherServlet` will look for a file in the path `WEB-INF/terrormovies-servlet.xml`, where the name `terrormovies-servlet` is picked up from the servlet-name value.

   Listing 4-2. web.xml snippets for setting up Spring MVC

   ```
   <servlet>
   <servlet-name>terrormovies</servlet-name>
    <servlet-class> org.springframework.web.servlet.DispatcherServlet </servlet-class>
    <load-on-startup>1</load-on-startup>
    </servlet>
   <servlet-mapping>
   <servlet-name>terrormovies</servlet-name>
   <url-pattern>/</url-pattern>
   </servlet-mapping>
   ```

2. You create a WEB-INF/terrormovies-servlet.xml file. This file will be the one Spring uses to configure the MVC support in the framework. You will define a minimal file here because that is all you need at the moment. You will create this file with the contents of Listing 4-3.

   Listing 4-3. terrormovies-servlet.xml file defining a Spring MVC configuration

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

```
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.1.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.1.xsd">
     <context:component-scan base-package="com.apress.pss.terrormovies" />
    <mvc:annotation-driven />
</beans>
```

3. Next you will define your first controller. For this first controller, you will focus on the "admin" part of the application. At the beginning, you will handle administration tasks with a particular URL namespace (`/admin/*`). That way, you can handle security using URL patterns. The controller can be found in Listing 4-4.

Listing 4-4. The Admin controller

```
package com.apress.pss.terrormovies.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/admin")
public class AdminController {

    @RequestMapping(method = RequestMethod.POST, value = "/movies")
    @ResponseBody
    public String createMovie(@RequestBody String movie) {
        System.out.println("Adding movie!! "+movie);
        return "created";
    }
}
```

Note that, for the purposes of web security, it doesn't really matter if you use a Spring MVC controller, like we do here, or if you use simple Servlets as we did in Chapter 2, or for that matter, if you use any other Servlet-based framework for developing your application. Remember that, at the core, the web part of Spring Security basically attaches itself to the standard Java Servlet Filter architecture. So if your application uses Servlets and Filters, you can leverage Spring Security's web support.

You can see that in this controller you specified the URL, the HTTP method, and the body of the request you will receive. The URL is built with a combination between the class-level `RequestMapping` annotation's value concatenated with the method-level `RequestMapping` annotation's value. So your URL will be `/admin/movies`. The HTTP method is specified as POST in the `RequestMapping` annotation on the method. The `RequestBody` annotation in the method parameter simply tells Spring to populate the movie parameter with the string that comes in the body of the request. And the `ResponseBody` annotation at the method level tells Spring to use the return value of the method as the body content of the response.

4. Now you will test your new functionality. First, make sure you restart the application by pressing Ctrl+C and `mvn jetty:run` again as you did before. You will be using curl for this first simple example. (Later, you will be using the user interface of the application itself.) Curl is a command-line tool you use to send HTTP requests to a server. It can be configured to use any HTTP method, add any arbitrary headers, use certificates, and so on. For testing your new functionality, you start your application (using `mvn jetty:run` as I explained in Chapter 2). Then, from the command line in another terminal, execute the following:

   ```
   curl -X POST -d "Die Hard" http://localhost:8080/admin/movies
   ```

You can see this in Figure 4-4.

```
Carlos-MacBook-Air:~ cscarioni$ curl -X POST -d "Die Hard" http://localhost:8080/admin/movies
createdCarlos-MacBook-Air:~ cscarioni$
```

Figure 4-4. *Executing curl to make a request*

If you look back in the first terminal where the application is running, you should see the text "Adding movie!! Die Hard" printed in the console as Figure 4-5 clearly shows. That is good. You are reaching the endpoint with your POST request to create movies. But in a real application context, maybe only an admin user should be allowed to create new movies on the system. Ensuring that is your next step.

```
2012-12-16 12:15:37.943:INFO:oejs.AbstractConnector:Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
Adding movie!! Die+Hard=
```

Figure 4-5. *A movie is created*

5. You have defined your first user and your first role. So you need an admin user. But what is an admin user? In this case, an admin user will simply be a user with a ROLE_ADMIN role. You need to decide where you are going to store your users and your roles. Because this is a greenfield fake application, you can decide to do whatever you want. For starters, you will do the easiest thing and store your users and roles in memory with the application. You will then modify this to a more realistic database-backed solution in later chapters. In later chapters, you will also fully explore some of the other providers you can use and even implement your own.

For defining your admin user and roles in memory, you will use the same user-service as in Chapter 2. This is a simple couple of XML elements you need to define in your configuration file applicationContext-security.xml, which you will create in the WEB-INF directory with the contents of Listing 4-5. The lines worth mentioning at this point are the ones that contain the element `<security:user-service>` and its child element `<security:user>`. Here, you can see you are defining your new admin user.

Listing 4-5. In-memory user detail configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
     http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
     http://www.springframework.org/schema/security
     http://www.springframework.org/schema/security/spring-security-3.1.xsd">

    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user authorities="ROLE_ADMIN" name="admin"
                    password="admin" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>

</beans>
```

6. The next step is to secure your URL, allowing access only to your newly defined admin user. You will make sure that only users with the ROLE_ADMIN role are able to reach your `AdminController`. To do this, you need to use the `<http>` element and the `<intercept-url>` element, as shown in Listing 4-6. You add those lines in the same file (applicationContext-security.xml) just after the opening `<beans>` element.

Listing 4-6. Securing URLs with role-based access

```
<security:http auto-config="true">
        <security:intercept-url pattern="/admin/**/*" access="ROLE_ADMIN" />
</security:http>
```

■ **Note** There is a catch in defining the security constraints of Spring Security. You cannot define them in the terrormovies-servlet.xml file, because they need to be loaded up with the Servlet listener and the filter chain. So they need to be in a proper `WebApplicationContext` defined with a Servlet listener, not the Servlet-related one. This is because the `DelegatingFilterProxy` will look for the root application context defined in the `ServletContext` that is loaded by the `ContextLoaderListener`.

If you define only a terrormovies-servlet.xml, because the filters load first, before the servlets, the filter won't be able to find any application context. So it won't be able to load up correctly. In short, you need to define your Spring Security configuration in one of the root application contexts.

---

7. Next you need to create the Servlet listener that will be aware of the configuration file applicationContext-security.xml. You also need to activate the Spring Security filter chain. To do that, you add the contents of Listing 4-7 to the web.xml file.

Listing 4-7. Servlet listener for loading the Spring Security configuration file

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
            /WEB-INF/applicationContext-security.xml
        </param-value>
    </context-param>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>

    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>
```

8. Next you try to access the URL again. This time, let's try the browser. Restart the application, and paste the following URL in your browser's address bar: `http://localhost:8080/admin/movies`.

This time, your browser should get redirected to the URL `http://localhost:8080/spring_security_login` and should show you the familiar login form from Chapter 1, which you can see in Figure 4-6.

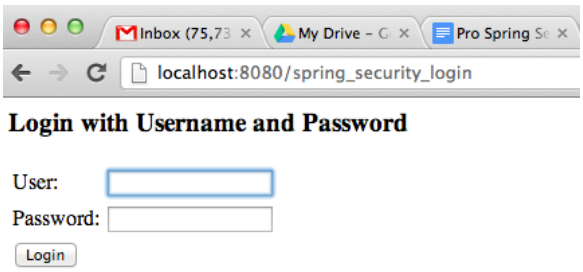


Figure 4-6. *Login form*

You will now look in more detail at what is going on in the application, and how the configuration you just defined is helping you secure the application. You will follow the request throughout its trip through the framework and look at the different steps that it takes.

When you defined the element `<http auto-config="true">` in your application context file, you got some configuration for free, including the most commonly required security filters. The request will go through all these filters in a predefined order.

When you make the HTTP request to the configured URL, and after your Servlet container (in this case, Jetty) deals with it, the request lands in the `DelegatingFilterProxy`, which in turn delegates the processing to the security `FilterChainProxy`.

Figure 4-7 shows each of the filters that your current request travels through in the framework.

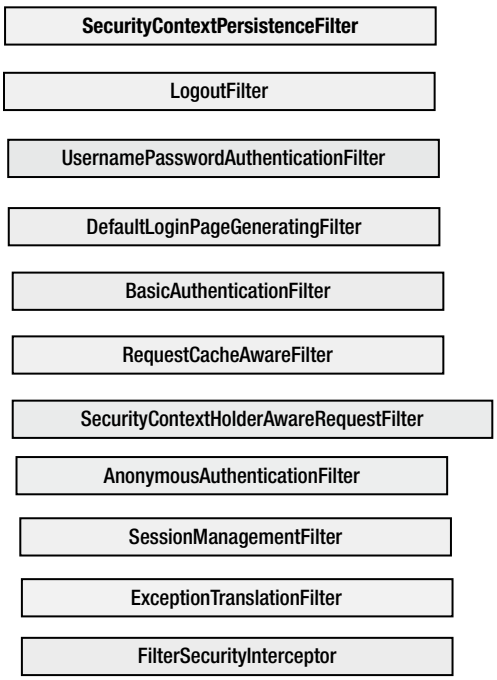


Figure 4-7. *The filter chain, with the filters invoked in the first request. They are invoked from the top down*

The first security filter that gets hit by the request is `SecurityContextPersistenceFilter`. When the request hits this filter, the framework tries to retrieve a `SecurityContext` from the standard Servlet session (`javax.servlet.http.HttpSession`). If there is not a context in the session, or the session itself is null (that is, it still doesn't exist because it hasn't been created yet), the framework creates a new empty `SecurityContext` (or, more accurately, an instance of its implementation class `SecurityContextImpl`). This is what happens in your current request. A new `SecurityContext` is created and associated to the current thread of execution. The request is then sent to the next filter in the chain. Figure 4-8 shows this interaction.

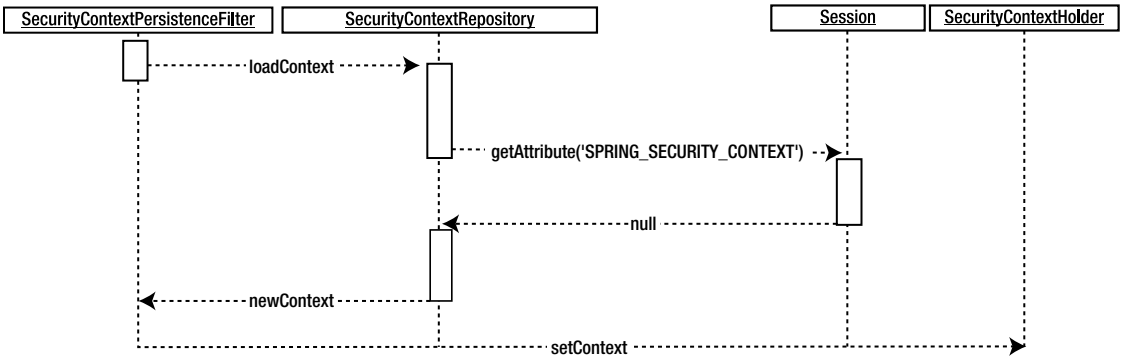


Figure 4-8. *Sequence diagram of the SecurityContextPersistenceFilter before forwarding the request to the next filter in the chain*

The next filter in the chain is the `LogoutFilter`. Because the request is not for the URL `/j_spring_security_logout`, the filter simply forwards the request to the next filter in the chain. This interaction is really simple and is shown in Figure 4-9.

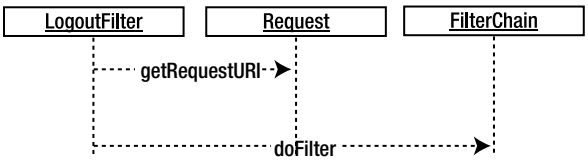


Figure 4-9. *LogoutFilter no-op interaction when the logout URL is not requested*

The request reaches the `UsernamePasswordAuthenticationFilter`. Because this request is not for the URL `/j_spring_security_check`, the filter simply forwards the request to the next filter in the chain. As with the previous filter, this interaction is simple and can be explained with the same diagram.

The next filter to run is the `DefaultLoginPageGeneratingFilter`. This filter will see that the request is not for the URL `/spring_security_login` and will forward the request to the next filter in the chain. As in the previous two filters, this interaction is simple and can be explained with the same diagram.

The next filter invoked is the `BasicAuthenticationFilter`. This filter looks for HTTP Basic Authentication headers in the request (the header "Authorization" with a value starting with "Basic"). Because none are found, the filter forwards the request to the next filter in the chain. As in the previous three filters, this interaction is simple and can be explained with the same simple diagram. The only difference is that instead of looking at the path of the request, this filter looks at the headers (`request.getHeader("Authorization")`).

The request arrives at the `RequestCacheAwareFilter` filter, which won't find any cached requests matching the current request. So it forwards the original request to the next filter. Cached requests are used normally when you do temporal redirections (like when you redirect to the login page) and then want to redirect back to the previously requested URL. Figure 4-10 shows this interaction.

Download from Wow! eBook <www.wowebook.com>

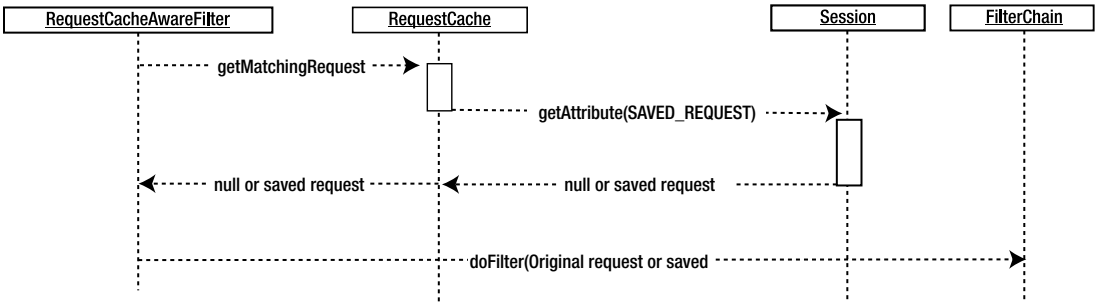


Figure 4-10. *RequestCacheAwareFilter tries to find a previously cached request or uses the original one*

The next filter invoked is `SecurityContextHolderAwareRequestFilter`. This filter wraps the request in a `SecurityContextHolderAwareRequestWrapper`, which implements the Servlet API security methods and forwards this request to the next filter in the chain.

The next filter in the chain is the `AnonymousAuthenticationFilter`. When your request hits this filter, the framework will see that there is no `Authentication` object currently set in the `SecurityContext`, and it creates an `AnonymousAuthenticationToken`. This object plays the role of an anonymous user in the system. As a matter of fact, it will have a username of `anonymousUser` and will be the granted authority ROLE_ANONYMOUS in the system. The request is then forwarded to the next filter in the chain. Figure 4-11 shows this interaction.

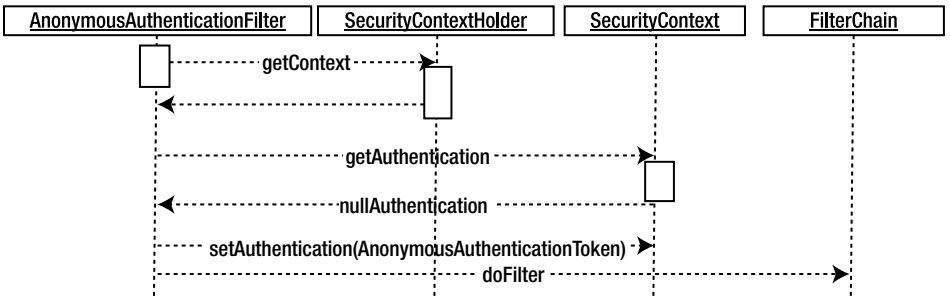


Figure 4-11. *AnonymousAuthenticationFilter at work; there is no authentication already set in the SecurityContext*

The request arrives at the `SessionManagementFilter`. At this moment, there is still no active Servlet session in the system for the particular agent that is accessing it. This filter checks that the current request is with an anonymous `Authentication` object (which was set by the previous filter in the context) and, finding that it is, it simply forwards the request to the next filter in the chain.

The next filter in the chain is `ExceptionTranslationFilter`. When the request arrives here, the only processing is to wrap the invocation to the next filter in the chain in a `try..catch` block. If any subsequent filter, or the request handler itself, throws an exception it will be caught by the `catch` block in this filter. This actually happens for the current request, as you will see when I describe the next filter. You see the interaction of this filter in Figure 4-12. You can see in the figure that when an exception is caught, the filter can invoke an instance of an implementation of `AuthenticationEntryPoint (as AuthenticationEntryPoint is an interface)`. An `AuthenticationEntryPoint` has the logic to start a new authentication process, and this normally means showing a login form to the logged-in user.

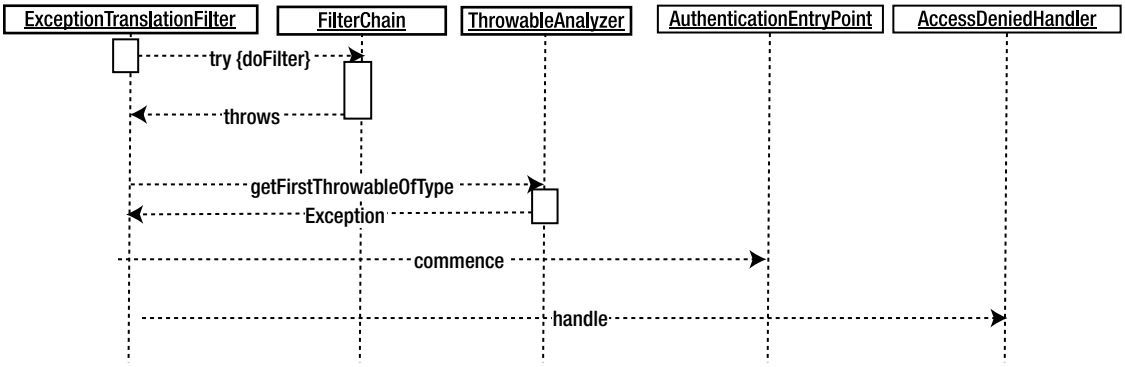


Figure 4-12. *ExceptionTranslationFilter wraps the invocation of the next filter in the chain with a try-catch block so that it can handle any security exception*

You finally arrive at the last filter in the default-configured chain shown in Figure 4-7. It is `FilterSecurityInterceptor`. I talked a lot about this filter in previous chapters, but I will refresh the information here in the context of our request.

When the request arrives in this filter, the filter creates a new `FilterInvocation` object that contains the request. This object is passed by the filter to the core `AbstractSecurityInterceptor` (from which `FilterSecurityInterceptor` extends). The interceptor (with some helpers' help) extracts the requested path from the request and checks whether it matches any of the patterns defined in the `<security:intercept-url pattern="" "access=""/>` pattern attribute. If it matches, it retrieves the required authorities from the "access" attribute contained in the `<intercept-url>` element. In this case study, it finds the value ROLE_ADMIN. The interceptor also extracts the `Authentication` object from the Security Context and sends these two elements (the `Authentication` object and the required authorities or `config` attributes) to the `AffirmativeBased` access-decision manager, which in turn sends them to the configured `RoleVoter` instance. The `RoleVoter` extracts the authorities associated with the `Authentication` object (ROLE_ANONYMOUS, in this case) and tries to match it to the received `ConfigAttribute`(s) (ROLE_ADMIN). The voter will not find any matches, so it will vote to deny access to the resource.

When the `AffirmativeBased AccessDecisionManager` receives the votes, it will see the deny vote and throw an `AccessDeniedException`. This exception bubbles all the way up to the `catch` block in the `ExceptionTranslationFilter`. The `catch` block in this filter will see that the current `Authentication` object stored in the `SecurityContext` is an Anonymous authentication and will start a new authentication process to obtain a complete Authentication from a user. This authentication process is handled by the class `LoginUrlAuthenticationEntryPoint`. This class builds a URL (`/spring_security_login` by default) and sends a redirect response to that URL using the standard Servlet redirect method `response.sendRedirect(redirectUrl)`. Figure 4-13 shows the work of the `FilterSecurityInterceptor` I just explained.

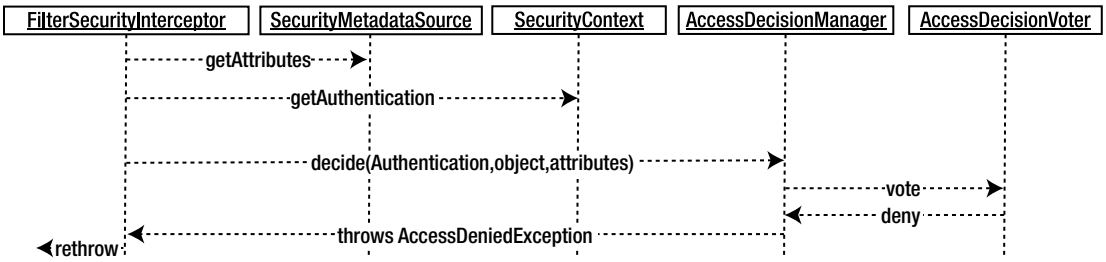


Figure 4-13. *FilterSecurityInterceptor interaction decision process. Access is denied in this case*

When the browser is redirecting and asks for the URL `/spring_security_login`, the following occurs:

- The process will be the same as for the first request until it reaches the `DefaultLoginPageGeneratingFilter`. At this point, the filter detects that the request is for the URL `/spring_security_login` and writes the login form's HTML data directly in the response object. Then the response will be rendered.

Now try to log in with incorrect credentials. We'll follow the request through the framework to see what happens:

1. In the login form, type the username **user** and the password **uspass**.
2. When the form is submitted, the filters are activated again in the same order as before. This time, however, when the request arrives at the `UsernamePasswordAuthenticationFilter`, the filter checks whether the request is for the URL `/j_spring_security_check` and sees that this is indeed the case. The filter extracts the username and password authentication information from the HTTP request parameters `j_username` and `j_password`, respectively. With this information, it creates the `UsernamePasswordAuthenticationToken` `Authentication` object, which then sends it to the `AuthenticationManager` (or more exactly, its default implementation `ProviderManager`) for authentication.
3. The `DaoAuthenticationProvider` gets called from the `ProviderManager` with the `Authentication` object. The `DaoAuthentication` provider is an implementation of `AuthenticationProvider`, which uses a strategy of `UserDetailsService` to retrieve the users from whichever storage they live in. With the configuration you currently have, it will try to find a user with the username "user" using the configured `InMemoryUserDetailsManager` (the implementation of `UserDetailsService` that maintains an in-memory user storage in a `java.util.Map`). Because there is no user with this username, the provider throws a `UsernameNotFoundException` exception.
4. The provider itself catches this exception and converts it into a `BadCredentialsException` to hide the fact that there is no such user in the application; instead; it treats the error as a common username-password combination error.
5. The exception will be caught by the `UsernamePasswordAuthenticationFilter`. This filter delegates to an instance of an implementation of `AuthenticationFailureHandler`, which in turn decides to redirect the response to the URL `/spring_security_login?login_error`. This way, the login form is shown again in the browser with an error message displayed. You can see this interaction in Figure 4-14.

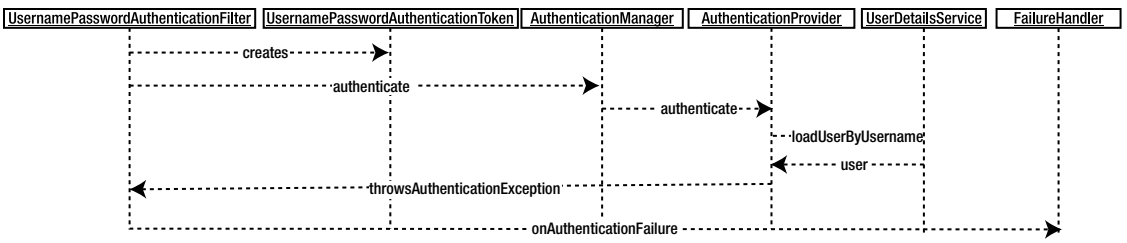


Figure 4-14. *Authentication filter when authentication details are incorrect*

Let's now log in with correct credentials:

1. First, let's create a new endpoint in the controller to retrieve some simple text. In the controller `AdminController`, create the method from Listing 4-8. Notice in the listing how you are using the GET HTTP method to handle requests. Then restart the application.

   Listing 4-8. A simple endpoint method that returns a simple string

   ```
   @RequestMapping(method = RequestMethod.GET, value = "/movies")
       @ResponseBody
       public String createMovie() {
           return "movie x";
       }
   ```

2. Go back to the URL `http:/localhost:8080/admin/movies`, and type **admin** as the username and **admin** as the password in the form. Then click the Login button.

   a. The request follows the same filter journey as before. This time, `InMemoryUserDetailsManager` finds a user with the requested username and returns that to `DaoAuthenticationProvider`, which creates a successful `Authentication` object.

   b. After successful authentication, the `UsernamePasswordAuthenticationFilter` delegates to an instance of `SavedRequestAwareAuthenticationSuccessHandler`, which looks for the original requested URL (`/admin/movies`) in the session and redirects the response to that URL.

3. When `http://localhost:8080/admin/movies` is requested, the request works its way through the filter chain as in the previous cases. This time, though, you already have a fully authenticated entity in the system. The request arrives in the `FilterSecurityInterceptor`.

   a. The `FilterSecurityInterceptor` receives an access request to the URL `/admin/movies`. Then it recovers the necessary credentials to access that URL (ROLE_ADMIN).

   b. The `AffirmativeBased` access-decision manager gets called, and in turn calls the `RoleVoter` voter. The voter evaluates the list of authorities of the authenticated entity and compares them with the required credentials to access the resource. Because the voter finds a match (ROLE_ADMIN is in both the Authentication authorities and the resource's `config` attributes), it votes with an ACCESS_GRANTED vote.

   c. The `FilterSecurityInterceptor` forwards the request to the next element in the request-handling chain, which in this case is Spring's `DispatcherServlet`.

   d. The request gets to the `AdminController`, which simply returns the string `movie x`, which then gets rendered to the browser. Figure 4-15 shows this result.

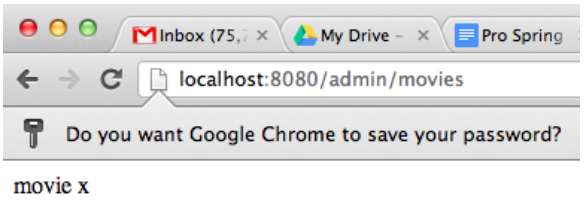


Figure 4-15. *movie x returned when accessing with correct credentials*

e. This is the complete flow of the Authentication and Authorization process. Figure 4-16 shows this full interaction in a pseudo flow chart.

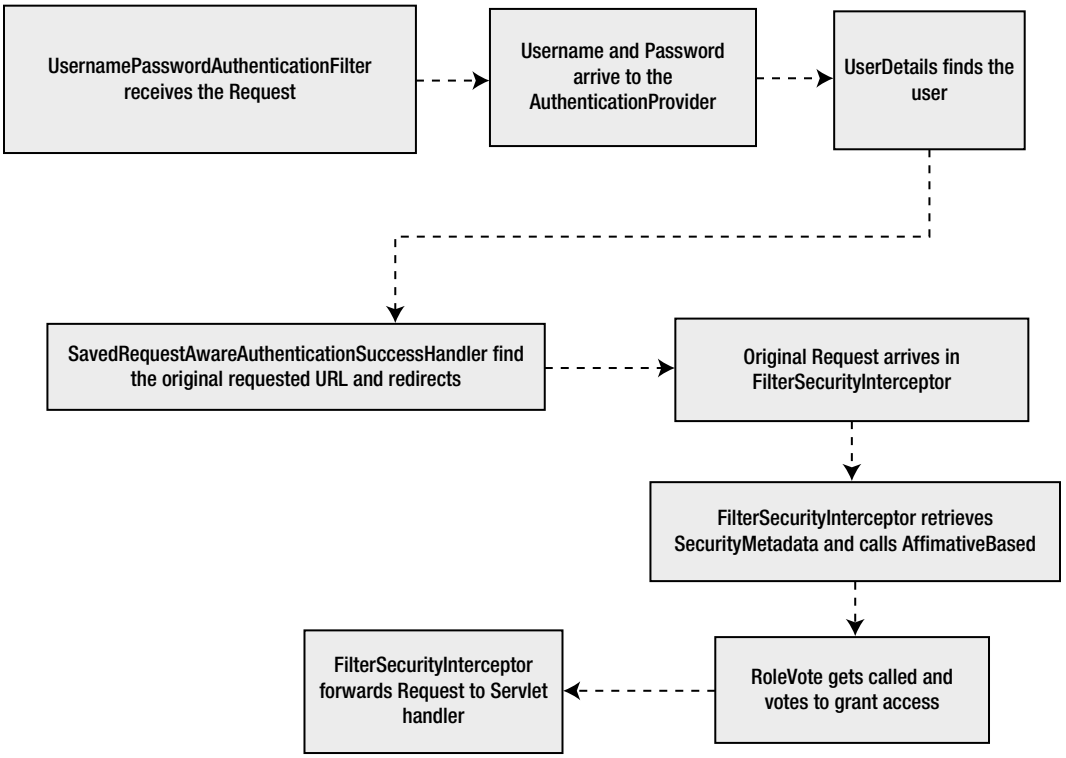


Figure 4-16. *Overall flow of a successful authentication and authorization process*

The Special URLs

From the preceding explanation, you can see that Spring Security's support for web security defines a few preconfigured URLs for you to use in your application. These URLs get special treatment in the framework. The main ones are the following:

- `/j_spring_security_check` This URL is used by the framework to determine that the incoming request is sending authentication information and asking to check these credentials and authenticate correspondingly.

- `/j_spring_security_logout` This URL is used by the framework to log out the currently logged-in user, invalidating the corresponding session and SecurityContext.
- `/spring_security_login` This is the URL that Spring Security uses to show the login form for the application. The framework will redirect to this URL when an authentication is needed but doesn't exist yet.

From the previous URLs, the first thing that comes to mind is how to configure your own Login form in the application and, in general, how to customize the login process instead of using the default one. That is what we'll do next.

Custom Login Form

When you configure the `<http>` element as you did before, Spring Security takes care of setting up a default login process for you, including a login URL, login form, default URL after login, and some other options. Basically, when Spring Security's context starts to load up, it will find that there is no custom login page URL configured, so it will assume the default one and create a new instance of `DefaultLoginPageGeneratingFilter` that will be added to the filter chain. As you saw before, this filter is the one that generates the login form for you.

If you want to configure your own form, you need to do the following. The first thing is to tell the framework to replace the default handling with your own. You define the following XML element as a child of the `<http>` element in the file applicationContext-security.xml:

```
<security:form-login login-page="/custom_login" />
```

This element tells Spring Security to change its default login-handling mechanism on startup. First, the `DefaultLoginPageGeneratingFilter` will no longer be instantiated. Let's try this first configuration out. With the new configuration in place, restart the application and try to access the URL `http://localhost:8080/admin/movies`.

You get redirected to the URL `/custom_login` and get a 404 HTTP error because you haven't defined any handler for this URL yet. This 404 page is shown in Figure 4-17.

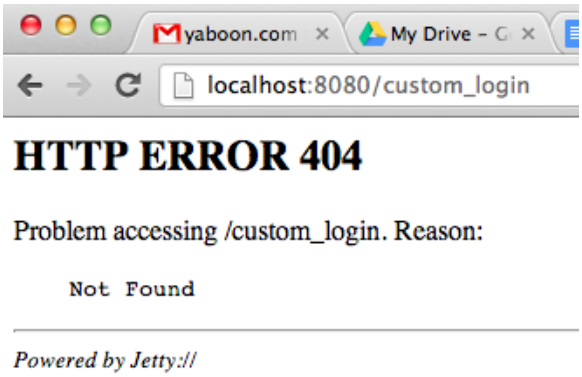


Figure 4-17. *Error 404 that appears when defining a new login handler page*

Let's create a `LoginController` controller next to the `AdminController` in the project. It should look something like Listing 4-9.

Listing 4-9. LoginController that handles the /custom_login URL specified in the configuration file

```
package com.apress.pss.terrormovies.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("")
public class LoginController {
    @RequestMapping(method = RequestMethod.GET, value = "/custom_login")
    public String showLogin(){
        return "login";
    }

}
```

The last line in the `showLogin` method returns a "logical view name," in Spring MVC parlance. This will be the name of the JSP file that has your login form. Now you create the fantastically designed `login.jsp` from Listing 4-10 in the folder WEB-INF/views in your application. You also add the XML snippet from Listing 4-11 into the terrormovies-servlet.xml file.

Listing 4-10. custom login.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Terror movies</title>
</head>
<body>
    <form action="/j_spring_security_check"  method="post">
        Username<input type="text" name="j_username"/><br/>
        Password<input type="text" name="j_password"/><br/>
        <input type="submit"/>
    </form>
</body>
</html>
```

Listing 4-11. View resolver definition

```
    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name = "prefix" value="/WEB-INF/views/" />
        <property name = "suffix" value=".jsp" />
    </bean>
```

If you restart the application and again go to the http://localhost:8080/admin/movies URL, you should see your new login form when you get redirected to the /custom_login URL. The form is shown in Figure 4-18. If you type **admin** as both the username and password, you get access to the movie x page as you did before with the default login form.

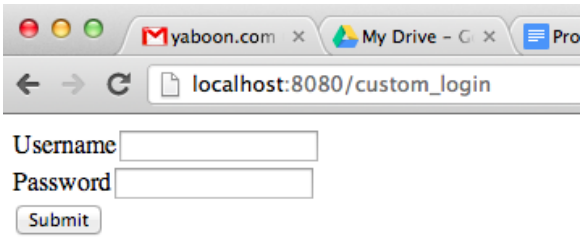


Figure 4-18. *Custom login form*

If you take a look at the login.jsp, you can see certain names for the username field, password field, and action attribute of the form element. These are not random names. Spring Security expects the use of these particular names in order to treat the authentication process correctly. Also, the form should use POST for sending the information to the server because this is required by the framework.

The element <form-login> supports many more configuration options, including changing the default j_username and j_password names for the authentication request parameters. To try it out, replace the current <form-login> element with the following one: <security:form-login login-page="/custom_login" username-parameter="user_param" password-parameter="pass_param"/>. Next, change the corresponding text fields in the login.jsp page to be named user_param and pass_param instead of j_username and j_password, respectively. Restart the application, go to the URL /admin/movies, and log in with **admin**/**admin**. You should be able to access the application without any problem.

The other attributes you can configure in the <form-login> element are these:

- default-target-url This attribute determines the URL that the logged-in user will be redirected to after successfully logging in, if he hasn't requested an explicit URL that triggered the authentication process. By default, this URL is the root of the application.
- authentication-failure-url This attribute specifies the URL that will be used to redirect to in case of a failed login attempt. In the default Spring Security configuration, this URL is /spring_security_login?login_error. This is the same as the normal login URL with a parameter of "login_error" appended to it. You can do something similar with your custom login page.

Let's give this attribute the value /custom_login?error. Then, in your login.jsp, let's add the content from Listing 4-12 just after the <body> tag.

Listing 4-12. Snippet showing an error in the login.jsp

```
<% if(request.getParameter("error") != null){
        out.println("ERROR LOGIN");
    }
%>
```

If you now restart the application and try to access the URL http://localhost:8080/admin/movies and use an incorrect username and password, you will get the Login page again, but with the error message shown at the top. Look at Figure 4-19 for the page you should be getting.

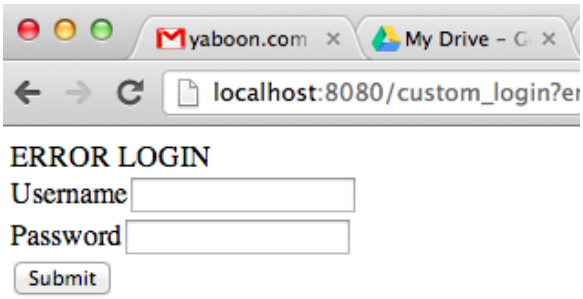


Figure 4-19. *A custom error shown in the custom form*

Note that this URL could be a different URL altogether, not related to the login URL at all. But the common pattern is to allow the user another attempt at login, showing her any errors.

- `authentication-success-handler-ref` Reference to an `AuthenticationSuccessHandler` bean in the Spring application context. This bean will be called on successful authentication and should handle the next step after authentication, usually deciding the redirect destination in the application. A current implementation in the form of `SavedRequestAwareAuthenticationSuccessHandler` takes care of redirecting the logged-in user to the original requested URL after successful authentication.
- `authentication-failure-handler-ref` Reference to an `AuthenticationFailureHandler` bean in the Spring application context. It is used to handle failed authentication requests. When an authentication fails, this handler gets called. A standard behavior for this handler is to present the login screen again or return a 401 HTTP status error. This behavior is provided by the concrete class `SimpleUrlAuthenticationFailureHandler`.

Let's develop a simple example implementation of the `AuthenticationFailureHandler interface`. It will simply return a 500 status code when failing to authenticate.

Create the class `ServerErrorFailureHandler` from Listing 4-13 in the package `com.apress.pss.terrormovies.security`.

Listing 4-13. AuthenticationFailureHandler implementation. ServerErrorFailureHandler

```
package com.apress.pss.terrormovies.security;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.authentication.AuthenticationFailureHandler;

public class ServerErrorFailureHandler implements AuthenticationFailureHandler{
```

```
    public void onAuthenticationFailure(HttpServletRequest request,
            HttpServletResponse response, AuthenticationException exception)
            throws IOException, ServletException {
        response.sendError(500);

    }
}
```

Then, in the applicationContext-security.xml file, replace the `<form-login>` element with the following:

```
<security:form-login login-page="/custom_login" authentication-failure-handler-
ref="serverErrorHandler"
    username-parameter="user_param" password-parameter="pass_param"/>.
```

And define the following Spring `<bean>` somewhere in that same file:

```
<bean id="serverErrorHandler" class="com.apress.pss.terrormovies.security.
ServerErrorFailureHandler"/>.
```

Restart the application, go to http://localhost:8080/admin/movies URL, use a random username and password, and click the submit button. You should get a 500 error in the browser as Figure 4-20 shows.



Figure 4-20. *Error 500 produced by custom AuthenticationFailureHandler*

Basic HTTP Authentication

Sometimes, you can't really use a login form for authenticating users. For instance, if your application is meant to be called by other systems instead of a human user, it doesn't make sense to show a login form to the other application. This is a pretty common use case. Web services talk to each other without user interaction, ESB systems integrate systems with one another, and JMS clients produce and consume messages from other systems.

In the context of HTTP-exposed interfaces that require no human user to access them, a common approach is to use HTTP basic authentication headers. HTTP authentication headers allow you to embed the security information (username and password) in the header of the request that you send to the server, instead of sending it in the body of the request, as is the case for the login form authentication.

HTTP uses a standard header for carrying this information. The header is appropriately named "Authorization." When using this header, the client that is sending the request (for example, a browser) concatenates the username and the password with a colon between them and then Base64 encodes the resulting string, sending the result of this

Download from Wow! eBook <www.wowebook.com>

in the header. For example, if you use the username **bart** and the password **simpson**, the client creates the string `bart:simpson` and encodes it prior to sending it in the header.

Let's use Basic HTTP authentication in our application. The first and only thing you need to do is replace the `<login-form>` element in your configuration file applicationContext-security.xml with the following one: `<security:http-basic/>`. After replacing it, you restart the application and go to the URL `http://localhost:8080/admin/movies` in the browser. A standard HTTP authentication box pops up asking you for your authentication details, as Figure 4-21 shows. Type **admin** as the username and password, and send the request. You successfully arrive in the `movie x` page that you already saw a couple of times before.



Figure 4-21. *Standard HTTP authentication form. Basic Authentication configuration*

When you use the `<http-basic/>` configuration element, Spring `Security's BasicAuthenticationFilter` comes into action. A `BasicAuthenticationEntryPoint` strategy will be configured into the `ExceptionTranslationFilter` on startup. When you make the first request to the URL `/admin/movies`, the framework behaves as before, throwing an access-denied exception that is handled by the `ExceptionTranslationFilter`. This filter delegates to a particular implementation strategy of `AuthenticationEntryPoint`—in this case, `BasicAuthenticationEntryPoint`. `BasicAuthenticationEntryPoint` adds the header "WWW-Authenticate: Basic realm="Spring Security Application"" to the response and then sends an HTTP status of 401 (Unauthorized) to the client. The client should know how to handle this code and work accordingly. (In the case of a browser, it simply shows the authentication pop up.)

When you introduce the username and password and submit the request, the request again follows the filter chain until it reaches the `BasicAuthenticationFilter`. This filter checks the request headers, looking for the "Authorization" header starting with "Basic." The filter extracts the content of the header and uses `Base64.decode` to decode the string, and then it extracts the username and password. The filter creates a `UsernamePasswordAuthenticationToken` object and sends it to the authentication manager for authentication in the standard way. The authentication manager will ask the authentication provider to retrieve the user and then create an `Authentication` object with it. This process is standard and independent of using `Basic Authentication` or form authentication.

Digest Authentication

Digest Authentication is a very close sibling of Basic HTTP Authentication. Its main purpose is to avoid sending clear text passwords on the wire, as Basic Authentication does, by hashing the password prior to sending it to the server. This makes Digest Authentication more complex than Basic Authentication.

Digest Authentication works with HTTP headers the same way that Basic Authentication does.

Digest Authentication is based in the use of a nonce for hashing the passwords. A *nonce* is an arbitrary server-generated number that is used in the authentication process and that is used only once. It is passed through the digest computation together with the username, password, nonce, URI being requested, and so on.

In the authentication process, both the server and client do the digest computation and they should match.

The main processing lies in two classes: `DigestAuthenticationFilter` and `DigestAuthenticationEntryPoint`.

`DigestAuthenticationFilter` queries the request's headers looking for the Authorization header, and then it checks that the header's value starts with "Digest." If this is the case, the request is carrying the security credentials that will be used for authentication.

`DigestAuthenticationEntryPoint` is the class that is invoked to generate a response that demands that a digest security authentication process begin. This class sets the header "WWW-Authenticate" with the correct values (including the nonce) so that the client agent (the browser) knows it has to start the digest authentication process.

To configure it, let's add the filter to the filter chain. In this case, there is no custom XML element to define it, so you need to create a `<custom-filter>` element and then create `<bean>`s for both the filter and the entry point. The new `<custom-filter>` element should look like this: `<security:custom-filter ref="digestFilter" before="BASIC_AUTH_FILTER"/>`. Also, it needs to be added as a child of the `<http>` element. In the `<http>` element, you add the attribute `entry-point-ref="digestEntryPoint"`, which will allow the `ExceptionTranslationFilter` to use this entry point when an `AccessDeniedException` is encountered. Listing 4-14 shows the two new beans you need to define in the file applicationContext-security.xml. You also need to give the ID "userService" to the configured `<user-service>` in the same file.

Listing 4-14. DigestAuthenticationFilter and DigestAuthenticationEntryPoint bean definitions

```
<bean id="digestFilter" class=
 "org.springframework.security.web.authentication.www.DigestAuthenticationFilter">
   <property name="userDetailsService" ref="userService" />
   <property name="authenticationEntryPoint" ref="digestEntryPoint" />
</bean>

<bean id="digestEntryPoint" class=
 "org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint">
   <property name="realmName" value="terrormovies-realm"/>
   <property name="key" value="terrorific" />
</bean>
```

If you restart the application and try to go to the URL `http://localhost:8080/admin/movies`, you will be presented with a browser dialog box asking for a username and password exactly like the one that was shown for Basic Authentication. This was the `DigestAuthenticationEntryPoint`'s work. As I explained before, the entry point will fill the response object with the required headers so that the browser knows it needs to show the login form. Log in with a username and password of **admin**, and you should be able to access the requested URL.

The browser will create its own digested message with the password input included and put it in the header. It will also put the rest of the information—namely, nonce, cnonce, realm, and so on—in the "Digest" header. An example "Digest" header that is sent to the server with your current request is the following:

```
'Digest username="admin", realm="terrormovies-realm",
nonce="MTM1NTY3NDc3NDIy....==", uri="/admin/movies",
response="225ea6fbad618cfdf1da7d4f7efe53b8", qop=auth,
nc=00000002, cnonce="376a9b27621880bd"'
```

When the request reaches `DigestAuthenticationFilter`, the headers of the request contain the required digest authentication header. The information in this header arrives as a csv string containing all the required information as I showed you in the last paragraph, including the nonce and the client nonce (`cnonce`). (A nonce is an arbitrary number used only once in a cryptographic communication. See `http://en.wikipedia.org/wiki/Cryptographic_nonce`.). The filter extracts the information from the header, retrieves the user from the `UserDetailsService`, and then computes the digest with the password from the retrieved user to see if the digest matches the one sent in the header by the client. If they match, access is granted.

Remember-Me Authentication

The remember-me authentication functionality is used for allowing returning users of the application to use it without needing to log in every time. Basically, the application will remember certain visitors, allowing them to just open the application and be greeted with their personalized version of the application, as if they were logged in.

Remember-me functionality is very convenient for users; however, it is also very dangerous and recommended for private (from home) use only.

The problem should be obvious. If you use an application from a public computer and this application remembers your profile information, the next person who accesses that application from that computer will be able to impersonate you with minimum effort.

It is also common practice to offer just a limited amount of functionality in the remember-me session. This means that even if you are logged-in automatically thanks to the remember-me functionality, you won't have access to the whole functionality of the application. More sensitive parts of the application might require you to formally log in to use them.

This is the case, for example, with `Amazon.com`. When you visit `Amazon.com` and log in, the next time you visit Amazon, the site will remember you, your recommendations, your name, and other information about you. But if you want to buy something, it will ask you to log in fully to access that functionality.

Remember-me authentication is typically supported by sending a cookie to the browser, which then, on subsequent sessions in the application, will be sent back to the server for auto login.

How does remember-me functionality work in Spring Security?

Remember-me functionality in Spring Security is supported mainly by two components: the `RememberMeServices` interface and the `RememberMeAuthenticationFilter` class. Let's see how they work in the context of a request.

In the application, replace the `<http-basic>` element with the `<form-login>` from the Basic and Digest examples.

Also, remember-me is not enabled by default. To enable it, include the following element inside the `<http>` element in the configuration file `<security:remember-me key="terror-key"/>`.

Now restart the application.

When the application starts up, the `RememberMeAuthenticationFilter` will be in the filter chain of the server. Also, a `TokenBasedRememberMeServices` will be instantiated and injected into the `AbstractAuthenticationProcessingFilter` replacing the no-op `NullRememberMeServices`.

Go and visit the URL `http://localhost:8080/admin/movies`, and log in with **admin** as the username and password.

When the request gets into the application, `UsernamePasswordAuthenticationFilter` (a subclass of `AbstractAuthenticationProcessingFilter`) will handle the authentication process in the standard way already explained.

After the authentication is successful, `UsernamePasswordAuthenticationFilter` invokes the configured `TokenBasedRememberMeServices's loginSuccess` method. This method looks to see if the request contains the parameter `_spring_security_remember_me` in order to apply the remember-me functionality. (If the property `alwaysRemember` is set to `true` in the service, it will also apply the remember-me functionality.) Because you didn't send this request, nothing will happen.

So let's add the parameter to the login form you have. Open the file `login.jsp` and somewhere inside the `<form>` paste the following element: `<input type="checkbox" name="_spring_security_ remember_me" value="yes"/>`. (As a value attribute, you can use any of the following values: "yes", "on", "1", or "true".)

Restart the application, and visit the URL `http://localhost:8080/admin/movies`. You should now see a check box along with the username and password fields. Select the check box, and log in with **admin**/**admin**.

This time, the request carries the required parameter and `TokenBasedRememberMeServices` does its work. It extracts the username and password from the `Authentication` object and creates a token with this information and a time to expire. It basically concatenates these three values and the remember-me key specified in the XML element (terror-key). And it creates an MD5 encoding out of the resulting string. This value will then be Base64-encoded again, together with the username, and added to the response as a cookie with the name `SPRING_SECURITY_REMEMBER_ME_COOKIE` that will be returned to the browser. You can see this cookie in Figure 4-22.

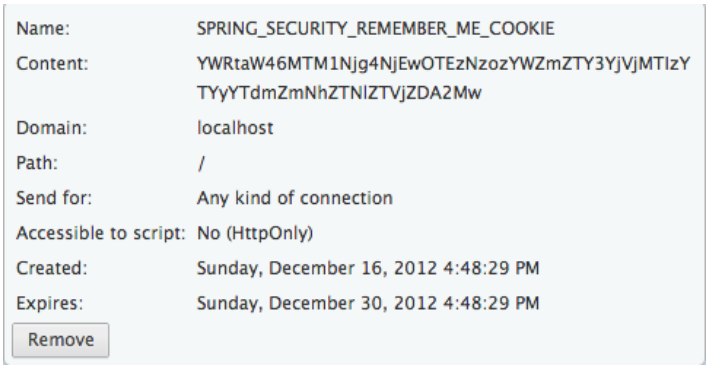


Figure 4-22. *Remember-me cookie example*

Restart the application. Visit the URL `http://localhost:8080/admin/movies`. You should be able to access the page without logging in.

When this request gets in the system, it is intercepted by the `RememberMeAuthenticationFilter`, which gets into action. The first thing the filter does is check that there is no current `Authentication` in the `SecurityContext`. Because this means there is no user logged in, the filter calls the `RememberMeServices`'s `autoLogin` method.

In the standard configuration, the `TokenBasedRememberMeServices` is the concrete class that implements `RememberMeServices`. This implementation's `autoLogin` method tries to parse the incoming cookie into its composing elements, which are the username, the hashed value of the combined elements (username + ":" + tokenExpiryTime + ":" + password + ":" + key), and the expiry time of the token. Then it retrieves the `UserDetails` from the `UserDetailsService` with the username, recomputes the hashed value with the retrieved user, and compares it with the arriving one. If they don't match, an `InvalidCookieException` is thrown. If they do match, the `UserDetails` is checked and an `Authentication` object is created and returned to the caller.

The `autoLogin` method extracts the remember-me cookie out of the request, decodes it, does some validation and then calls the configured `UserDetailsService`'s `loadUserByUsername` method with the username extracted from the cookie. It then creates a `RememberMeAuthenticationToken` object (an implementation of `Authentication`).

The `RememberMeAuthenticationFilter` then tries to authenticate this new `Authentication` object against the `AuthenticationProvider`'s implementation of `RememberMeAuthenticationProvider`, which simply returns the same `Authentication` object after making sure that the hash from the incoming request matches the stored one for the remember-me key.

This `Authentication` object will be used by the `Security Interceptor` to allow access to the requested URL.

Allowing Remember-Me Access to Selected Parts of the Application

Remember-me authentication can be easily configured so that certain URLs require a fully authenticated user (meaning the user is explicitly logged in) to access them. To do this in your application, replace the `<security:intercept-url/>` element you have in the file `applicationContext-security.xml` with the following one:

```
<security:intercept-url pattern="/admin/*" access="ROLE_ADMIN,IS_AUTHENTICATED_FULLY" />
```

You have added the access rule `IS_AUTHENTICATED_FULLY` to the `access` attribute.

You still need more configuration to make this work. By default, Spring Security's `<http>` element configures an `AffirmativeBased` access-decision manager. `AffirmativeBased`, as I explained in Chapter 3, grants access to a

resource if any of the configured voters votes to grant access to the resource. By default, both the `RoleVoter` and the `AuthenticatedVoter` are configured in the manager, and the `RoleVoter` is queried first. The `RoleVoter` will vote to grant access, so the `AuthenticatedVoter` won't be called at all.

You need to define a `UnanimousBased` access-decision manager in your Spring Security configuration and reference that one from the `<http>` element.

Let's add the bean definition from Listing 4-15 to your `applicationContext-security.xml`. And make the `<http>` element's opening tag look like the following:

```
<security:http auto-config="true" access-decision-manager-ref="accessDecisionManager">.
```

Listing 4-15. Unanimous AccessDecisionManager

```
<bean id="accessDecisionManager" class="org.springframework.security.access.vote.UnanimousBased">
        <constructor-arg>
          <list>
            <bean class="org.springframework.security.access.vote.RoleVoter"/>
            <bean class="org.springframework.security.access.vote.AuthenticatedVoter"/>
          </list>
        </constructor-arg>
</bean>
```

Restart the application, and visit the URL `http://localhost:8080/admin/movies`.

This time, the access-decision manager calls the `AuthenticatedVoter`. The `AuthenticatedVoter` will see that the URL requires a fully authenticated user to access it, so it will query the requesting `Authentication` to see if it is fully authenticated. The way this check takes place is straightforward: the voter simply checks if the `Authentication` object implementation is neither an `AnonymousAuthenticationToken` nor a `RememberMeAuthenticationToken` instance, assuming it is then a fully authenticated `Authentication` object and allowing access. In this case, access will be denied because the `Authentication` object is a `remember-me` implementation. This means that the login form will be shown even if a remember-me option was used previously.

Remember-me authentication supports the use of persistent storage, so the token is kept in a datastore and survives application restarts. The main class supporting persistent remember-me storage is `PersistentTokenBasedRememberMeServices`, which extends from `AbstractRememberMeServices` the same way that `TokenBasedRememberMeServices` does. To activate the use of the persistent functionality, you need to add the attribute `data-source-ref="someDataSource"` to the element `<security:remember-me>` that you defined in the configuration file. This way, you can reference a data source bean in the application context.

Another way to have persistent tokens is to define the attribute `token-repository-ref="someTokenRepo"` in the element <security:remember-me>. In this attribute you need to put a reference to a bean of type `PersistentTokenRepository`, or more exactly, an implementation of `PersistentTokenRepository` as that is an interface. If you were to use the implementation `JdbcTokenRepositoryImpl`, it would be as if you were defining the `data-source-ref` attribute that I explained before, because this is what such an attribute does internally. However, using the attribute `token-repository-ref`, you also have available the implementation `InMemoryTokenRepositoryImpl`, which is backed by a simple in-memory map and is recommended for testing purposes only. You could also create your own implementation based on some other kind of datastore and inject it into your `<remember-me>` element.

Persistent remember-me tokens come with a nice feature in the current implementation, which is detecting remember-me cookie thefts.

In the persistent model, tokens are stored against a `series-id`. A `series-id` is simply a random Base64 string that is generated whenever a successful login is done in the system. This `series-id` will be part of the remember-me cookie (SPRING_SECURITY_REMEMBER_ME_COOKIE) that is sent back to the browser together with the token when this successful login happens.

Every time a new `autoLogin` request comes to the `PersistentTokenBasedRememberMeServices`, the cookie values (`series-id` and `token`) are extracted and compared with the stored ones. If they both match, another token is generated for the same `serial-id` and the datastore is updated with this new token value. If the `series-id` matches but the token doesn't match, it is assumed that a cookie theft has happened (or basically two people have the same cookie). This is assumed because, as I just explained, the `series-id` is a quasi-unique random number that is generated on successful login and maintained for that user in the cookie. That means it is virtually impossible that some other user in a different browser will have the same `series-id`, unless he had the same cookie. This is because every time a remember-me autologin is performed, the `series-id` is used to retrieve the token from the store and then the token is updated.

On subsequent requests, if a user's token doesn't match the stored one, this is because someone else accessed the autologin functionality from somewhere else causing the token to be updated for the user's series-id, so that the old legitimate user token doesn't match the one that is now stored.

When the system detects this, it throws a `CookieTheftException` and removes all the user tokens from the datastore.

This implementation from Spring Security is based on the following article: `http://jaspan.com/improved_persistent_login_cookie_best_practice`.

Logging Out

Logging out is pretty simple. When you log out of an application, you want the application to end your current session, but also to remove any information it might have stored on the client for you.

In Spring Security, logging out is very easy. The only thing you need to do by default is to visit the URL `/j_spring_security_logout`. Let's try that. Remove the latest added `IS_AUTHENTICATED_FULLY` from the configuration XML file and restart the application.

Now go to the URL `http://localhost:8080/admin/movies` and log in with **admin**/**admin** again. Select the check box for activating `remember-me` functionality. You should be able to log in without problems.

Now if you look at the cookies stored in your browser, you should see two cookies for the localhost domain: `JSESSIONID and SPRING_SECURITY_REMEMBER_ME_COOKIE`. Figure 4-23 shows the two cookies. You would expect that if you log out, these two cookies disappear from the browser, basically removing any trace of the application from your browser. Let's do it.

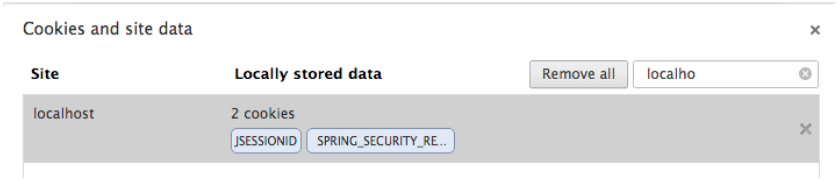


Figure 4-23. *Remember-me and session cookies*

Go and visit the URL `http://localhost:8080/j_spring_security_logout`. You should be logged out of the application. If you open the cookies of your browser, you will see that the cookie `SPRING_SECURITY_REMEMBER_ME_COOKIE` is gone. The `JSESSIONID` cookie exists, but the session was already invalidated by the framework.

The flow of the logout request is as follows. When the request arrives, it follows the filter chain until it arrives at the `LogoutFilter`. This filter notices that the URL that is being requested is for logout. The filter calls the configured `LogoutHandler(s)`, which in the running application are `SecurityContextLogoutHandler` and `TokenBasedRememberMeServices`. (They implement the `LogoutHandler` interface.)

The `SecurityContextLogoutHandler` invalidates the Servlet session, in the standard Servlet way, calling the `invalidate` method on the `HttpSession` object and also clearing the `SecurityContext` from Spring Security.

TokenBasedRememberMeServices simply removes the remember-me cookie by setting its age to 0.

Spring Security 3.1 includes a new LogoutHandler called CookieClearingLogoutHandler. This handler, as its name implies, removes cookies as specified in its constructor. This handler, however, is not configured by default. Let's see how it works. Let's remove the JSESSIONID cookie from the browser when logging out.

To enable the CookieClearingLogoutHandler handler, add the following XML element as a child of the <http> element in the applicationContext-security.xml file: <security:logout delete-cookies="JSESSIONID"/>.

After restarting the application, logging in, and logging out, you can go to the cookies section in your browser and you will see that the JSESSIONID cookie is no longer there. It has effectively been removed.

The LogoutFilter, after calling the LogoutHandler(s), calls the LogoutSuccessHandler's onLogoutSuccess method, which, in the default configuration, redirects to a target URL. By default, this URL is the root of the application. The target URL can be configured using either a request parameter or the referrer header from the request. Let's try this out.

Replace your current <logout> element in the applicationContext-security.xml file with the following one:

```
<security:logout delete-cookies="JSESSIONID" success-handler-ref="logoutRedirectToAny"/>
```

And somewhere else in the same file, define the following bean:

```
<bean id="logoutRedirectToAny" class="org.springframework.security.web.authentication.logout.
SimpleUrlLogoutSuccessHandler">
    <property name="targetUrlParameter" value="redirectTo"/>
</bean>
```

After this is done, go and restart the application, go to http:localhost:8080/admin/movies, log in, and then request the following URL:

```
/j_spring_security_logout?redirectTo=http://www.google.com
```

You will be logged out and redirected to Google.

The current implementation LogoutSuccessHandler (SimpleUrlLogoutSuccessHandler) extends from AbstractAuthenticationTargetUrlRequestHandler, which is also used by the authentication process to redirect to a determined URL after authenticating successfully. So both processes work in the same way. (Both the successful authentication and successful logout use a redirection strategy for request handling.) You can go ahead and try it out using the attribute authentication-success-handler-ref in the <form-login> element and ensuring that your login form sends the required parameter for the redirection.

The Session (javax.servlet.http.HttpSession) and the SecurityContext

Traditionally in Java web applications, user session information is managed with the HttpSession object.

In Spring Security, at a low level, this is still the case. However, as I mentioned before, Spring Security introduces some new concepts for handling user-session information.

In an application using Spring Security, you will rarely access the Session object directly for retrieving user details. Instead, you will use SecurityContext (and its implementation class) and SecurityContextHolder (and its implementing classes). The SecurityContextHolder allows quick access to the SecurityContext, the SecurityContext allows quick access to the Authentication object, and the Authentication object allows quick access to the user details.

Let's see how it works. Go ahead and modify the current AdminController to look like Listing 4-16. Remember not to remove the package declaration or the imports from the file. Also, add any extra needed imports.

Listing 4-16. AdminController accessing the SecurityContext and extracting user information

```
@Controller
@RequestMapping("/admin")
public class AdminController {

    @RequestMapping(method = RequestMethod.POST, value = "/movies")
    @ResponseBody
    public String createMovie(@RequestBody String movie) {
        System.out.println("Adding movie!! "+movie);
        return "created";
    }

    @RequestMapping(method = RequestMethod.GET, value = "/movies")
    @ResponseBody
    public String createMovie() {
        UserDetails user = (UserDetails)SecurityContextHolder.getContext().getAuthentication().
        getPrincipal();
        System.out.println("returned movie!");
        return "User "+user.getUsername()+" is accessing movie x";
    }
}
```

Restart the application, and go to `http:localhost:8080/admin/users`. Log in with **admin**/**admin**. You should get the message "User admin is accessing movie x" in the browser window, as shown in Figure 4-24. Let's make it more interesting and add more information to your Users. You will make your users have a last name and first name.

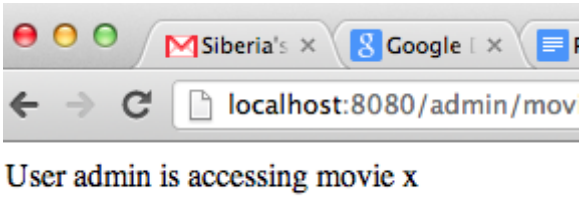


Figure 4-24. *A message with username information extracted from the Authentication object*

You want to keep the InMemory model of storing and retrieving users, but you need a more flexible user model. Our user model needs to implement directly or indirectly the `UserDetails` interface. So let's create the `User` class from Listing 4-17 in the package `com.apress.pss.terrormovies.model`. Next let's replace applicationContext-security.xml with the one from Listing 4-18 Finally, let's replace `AdminController` with the content from Listing 4-19.

Listing 4-17. Custom User class

```
package com.apress.pss.terrormovies.model;
import java.util.Collection;

import org.springframework.security.core.GrantedAuthority;

public class User extends org.springframework.security.core.userdetails.User{

    private String lastname;

    public User(String username, String password, boolean enabled,
            boolean accountNonExpired, boolean credentialsNonExpired,
            boolean accountNonLocked,
            Collection<? extends GrantedAuthority> authorities, String lastname) {
        super(username, password, enabled, accountNonExpired, credentialsNonExpired,
                accountNonLocked, authorities);
        this.lastname = lastname;
    }

public User(String username, String password, Collection<? extends GrantedAuthority>
authorities, String lastname) {
      this(username, password, true, true, true, true, authorities, lastname);
  }

public String getLastname() {
    return lastname;
}
}
```

Listing 4-18. applicationContext-security.xml with custom UserDetails

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true" access-decision-manager-ref="accessDecisionManager">
        <security:intercept-url pattern="/admin/*"
            access="ROLE_ADMIN" />
            <security:remember-me key="terror-key" />
            <security:logout delete-cookies="JSESSIONID" success-handler-ref="logoutRedirectToAny"/>
            <security:form-login  login-page="/custom_login"
                  authentication-failure-handler-ref="serverErrorHandler"
                  username-parameter="user_param" password-parameter="pass_param"/>
    </security:http>
```

```
    <bean id="accessDecisionManager" class="org.springframework.security.access.vote.
    UnanimousBased">
        <constructor-arg>
          <list>
            <bean class="org.springframework.security.access.vote.RoleVoter"/>
            <bean class="org.springframework.security.access.vote.AuthenticatedVoter"/>
          </list>
        </constructor-arg>
    </bean>

    <security:authentication-manager>
        <security:authentication-provider user-service-ref="inMemoryUserServiceWithCustomUser"/>
    </security:authentication-manager>

    <bean id="inMemoryUserServiceWithCustomUser"
           class="com.apress.pss.terrormovies.spring.CustomInMemoryUserDetailsManager">
        <constructor-arg>
        <list>
            <bean class="com.apress.pss.terrormovies.model.User">
                <constructor-arg value="admin"/>
                <constructor-arg value="admin"/>
                <constructor-arg>
                    <list>
                        <bean class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                          <constructor-arg value="ROLE_ADMIN"/>
                        </bean>
                    </list>
                </constructor-arg>
                <constructor-arg value="Scarioni"/>
            </bean>
        </list>
        </constructor-arg>
    </bean>

    <bean id="logoutRedirectToAny"

class="org.springframework.security.web.authentication.logout.SimpleUrlLogoutSuccessHandler">
        <property name="targetUrlParameter" value="redirectTo"/>
    </bean>
    <bean id="serverErrorHandler" class="com.apress.pss.terrormovies.security.
ServerErrorFailureHandler"/>
</beans>
```

Listing 4-19. AdminController using custom User and retrieving lastname

```
package com.apress.pss.terrormovies.controller;

import org.springframework.security.core.context.SecurityContextHolder;
import com.apress.pss.terrormovies.model.User;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
```

Download from Wow! eBook <www.wowebook.com>

```
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/admin")
public class AdminController {

    @RequestMapping(method = RequestMethod.POST, value = "/movies")
    @ResponseBody
    public String createMovie(@RequestBody String movie) {
        System.out.println("Adding movie!! "+movie);
        return "created";
    }

    @RequestMapping(method = RequestMethod.GET, value = "/movies")
    @ResponseBody
    public String createMovie() {
        User user = (User)SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        System.out.println("returned movie!");
        return "User "+user.getLastname()+" is accessing movie x";
    }
}
```

You also need to create your own `InMemory` user details service, because the default one will create Spring Security's own `User` instance and not your custom class instances. In the package `com.apress.pss.terrormovies.spring`, create the class `CustomInMemoryUserDetailsManager` from Listing 4-20.

Listing 4-20. CustomInMemoryUserDetails Manager class

```
package com.apress.pss.terrormovies.spring;

import java.util.Collection;
import java.util.HashMap;
import java.util.Map;

import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;

import com.apress.pss.terrormovies.model.User;

public class CustomInMemoryUserDetailsManager implements UserDetailsService {

    private Map<String, User> users = new HashMap<String, User>();

    public CustomInMemoryUserDetailsManager(Collection<User> users) {
        for (User user : users) {
            this.users.put(user.getUsername().toLowerCase(), user);
        }
    }
```

```
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        User user = users.get(username.toLowerCase());

        if (user == null) {
            throw new UsernameNotFoundException(username);
        }
        User userNew = new User(user.getUsername(), user.getPassword(), user.getAuthorities(),
        user.getLastname());

        return userNew;
    }

}
```

Let's restart the application, visit the URL http://localhost:8080/admin/movies, and log in. You can see the message "User Scarioni is accessing movie x" as Figure 4-25 shows.

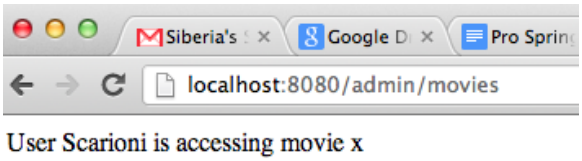


Figure 4-25. *Application using a custom User object and accessing the last name of the user*

Beyond Simple User Roles: Using Spring Expression Language to Secure the Web Layer

Spring Security 3 introduced support for SpEL into the framework. SpEL makes it possible to use programming expressions inside bean definitions and other parts of the Spring portfolio—for example, in some annotations you will see later, which allows you to manipulate objects at runtime. It basically allows the developer to embed code in the configuration files as simple strings and then evaluate those strings at runtime, very much like a dynamic language would allow you to.

Following is the current supported functionality of SpEL extracted directly from Spring documentation at http://static.springsource.org/spring/docs/3.0.x/reference/expressions.html:

- Literal expressions
- Boolean and relational operators
- Regular expressions
- Class expressions
- Accessing properties, arrays, lists, maps
- Method invocation
- Relational operators
- Assignment
- Calling constructors
- Bean references

- Array construction
- Inline lists
- Ternary operator
- Variables
- User-defined functions
- Collection projection
- Collection selection
- Templated expressions

Spring Security leverages this SpEL functionality in many ways. The first one we are going to look at is web-layer security using SpEL.

It should come as no surprise that Spring Security's main Web support for SpEL is configured using Servlet Filters. To activate SpEL support, you add the following attribute to the `<http>` element in the applicationContext-security.xml file: `use-expressions="true"`. When you do this, Spring Security changes its startup process a little bit. This time, when the Spring namespace parser mechanism is parsing the XML, it will notice that this attribute is in the `<http>` element. When creating the `FilterSecurityInterceptor` bean definition, it adds a definition of `ExpressionBasedFilterInvocationSecurityMetadataSource` to it. This final class will be used at startup to map URLs to SpEL parsed expressions.

In the default case, it will also add a `WebExpressionVoter` to the list of voters configured in the access-decision manager. So let's use it. Let's remove the custom access-decision manager bean from the configuration file, and let's also remove the corresponding reference from the `<http>` element (the attribute `access-decision-manager-ref`), and let it use the default one that is configured when none is explicitly set.

When using the expression support, now the value in the `access` attribute of the `<intercept-url>` element will be interpreted as a SpEL expression. (Remember that before it was either a role or a value like `IS_FULLY_AUTHENTICATED`.)

So let's add an expression to `<intercept-url>`. In the `access` attribute put the value `hasRole('ROLE_ADMIN')`.

Restart the application, go to `http://localhost:8080/admin/movies`, and log in with **admin**/**admin**. You should be able to access the page from Figure 4-25 without a problem.

The way the flow works is the following. Before and after logging in, the access-decision manager will make a call to the `WebExpressionVoter` to decide if it should grant access or not.

The `WebExpressionVoter` retrieves the attribute `hasRole('ROLE_ADMIN')` from the configuration for the requested URL. Then it creates a SpEL evaluation context using the `Authentication` object and the `FilterInvocation` object. An evaluation context is where references are resolved when encountered during expression evaluation. In this case, an instance of `WebSecurityExpressionRoot` is created and used as the root of the evaluation context. This means that methods called on the SpEL will evaluate against this root object. In other words, the `WebExpressionVoter` for this particular expression will call the method `hasRole` in an instance of the class `WebSecurityExpressionRoot`.

If you go to the source of the class `WebSecurityExpressionRoot` and into its class hierarchy, you will find all its available variables and methods for access through expressions. Namely, you will find the following:

- `request` Direct public access to the request object.
- `boolean hasIpAddress(String)` Evaluates whether the request matches a particular IP address or IP mask.
- `Authentication getAuthentication()` Access to the `Authentication` object.
- `boolean hasAuthority(String authority)` Alias for `hasRole`.
- `boolean hasAnyAuthority(String... authorities)` Alias for `hasAnyRole`.
- `boolean hasRole(String role)` Determines whether the `Authentication` object has the specified role.

- `boolean hasAnyRole(String... roles)` Determines whether the `Authentication` object has any of the specified roles.
- `boolean permitAll()` Returns `true` all the time.
- `boolean denyAll()` Returns `false` all the time.
- `boolean isAnonymous()` Determines whether the current Authentication is an anonymous one.
- `boolean isAuthenticated()` Determines whether the current Authentication is not an anonymous one.
- `boolean isRememberMe()` Determines whether the current Authentication is a remember-me one.
- `boolean isFullyAuthenticated()` Determines whether the current Authentication is neither a remember-me nor an anonymous one.
- `boolean hasPermission(Object target, Object permission)` Evaluates whether the current Authentication has certain permissions on the specified domain object. I'll say more about this in Chapter 7.
- `boolean hasPermission(Object targetId, String targetType, Object permission)` Evaluates whether the current Authentication has certain permissions on the specified domain object. I'll say more about this in Chapter 7.
- `Object getPrincipal()` Returns the principal from the authentication.

It is possible to create complex expressions combining more than one expression using "or" or "and". For example, in the current example, you could allow access only to users with a last name of "Scarioni" and coming from localhost (IP 127.0.0.1). You would do that with the following expression:

```
hasIpAddress('127.0.0.1') and (isAnonymous() ? false : principal.lastname == 'Scarioni')
```

Try that out. Restart the application, visit http://127.0.0.1:8080/admin/movies, and log in with **admin**/**admin.** You should be able to access the page without a problem.

Extend with Your Own Expressions

Although the default functionality offered by the expression-handling mechanism is rich, sometimes you might need to add more expressions that are not readily available. Suppose that you want to support an expression such as "over18." For that, you need to add your own `ExpressionHandler` to the configuration:

1. Define the following as a child element of the `<http>` element:

   ```
   <security:expression-handler ref="expressionHandler"/>
   ```

2. Alter the `User` class to add an "age" attribute as shown in Listing 4-21.

   Listing 4-21. User with the age attribute

   ```
   public class User extends org.springframework.security.core.userdetails.User{

       private String lastname;
       private int age;
   ```

```
    public User(String username, String password, boolean enabled,
            boolean accountNonExpired, boolean credentialsNonExpired,
            boolean accountNonLocked,
            Collection<? extends GrantedAuthority> authorities, String lastname, int age) {
        super(username, password, enabled, accountNonExpired, credentialsNonExpired,
                accountNonLocked, authorities);
        this.lastname = lastname;
        this.age=age;
    }

      public User(String username, String password,
            Collection<? extends GrantedAuthority> authorities, String lastname, int age) {
               this(username, password, true, true, true, true, authorities, lastname,age);
        }

    public String getLastname() {
        return lastname;
    }

    public int getAge() {
        return age;
    }

}
```

3. Create the class `CustomWebSecurityExpressionHandler`, as shown in Listing 4-22, in the package `com.apress.pss.terrormovies.security`. This class instantiates the `CustomWebSecurityExpressionRoot` you will create in the next step. This class is needed because the default one, `DefaultMethodSecurityExpressionHandler`, has hardcoded the construction of `MethodSecurityExpressionRoot`. And you need to use your custom `SecurityExpressionRoot`.

Listing 4-22. CustomWebSecurityExpressionHandler

```
public class CustomWebSecurityExpressionHandler extends AbstractSecurityExpressionHandler
<FilterInvocation> {

    @Override
    protected SecurityExpressionRoot createSecurityExpressionRoot(
            Authentication authentication, FilterInvocation invocation) {
        CustomWebSecurityExpressionRoot root =
              new CustomWebSecurityExpressionRoot(authentication, invocation);
        root.setPermissionEvaluator(getPermissionEvaluator());
        root.setTrustResolver(new AuthenticationTrustResolverImpl());
        return root;
    }

}
```

4. Create the class `CustomWebSecurityExpressionRoot` as shown in Listing 4-23 in the package `com.apress.pss.terrormovies.security`. This class is the one that will include the new method you want to make available to the SpEL expressions. The method will be `isOver18()`.

   Listing 4-23. CustomWebSecurityExpressionRoot

   ```
   public class CustomWebSecurityExpressionRoot extends WebSecurityExpressionRoot{

       public CustomWebSecurityExpressionRoot(Authentication a, FilterInvocation fi) {
           super(a, fi);
       }

       public boolean isOver18(){
           User user = (User)this.getPrincipal();
           return user.getAge() > 18;
       }

   }
   ```

5. Then create the following bean somewhere in the `applicationContext-security.xml` file:

   ```
   <bean id="expressionHandler"
         class="com.apress.pss.terrormovies.security.CustomWebSecurityExpressionHandler"/>
   ```

6. Add the new age constructor argument to the defined user, and complete the access expression in the `<intercept-url>` so that it uses the "over18" functionality. At the end, your applicationContext-security.xml file should look like Listing 4-24.

   Listing 4-24. applicationContext-security.xml with "over18" expression functionality

   ```
   <?xml version="1.0" encoding="UTF-8"?>
   <beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:security="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
       http://www.springframework.org/schema/security
       http://www.springframework.org/schema/security/spring-security-3.1.xsd">
       <security:http auto-config="true" use-expressions="true" >
           <security:expression-handler ref="expressionHandler"/>
           <security:intercept-url pattern="/admin/*"
               access="hasIpAddress('127.0.0.1')
                and (isAnonymous() ? false : principal.lastname == 'Scarioni') and over18" />
               <security:remember-me key="terror-key" />
               <security:logout delete-cookies="JSESSIONID"
                     success-handler-ref="logoutRedirectToAny"/>
               <security:form-login  login-page="/custom_login"
                      authentication-failure-handler-ref="serverErrorHandler"
                      username-parameter="user_param" password-parameter="pass_param"/>
       </security:http>
   ```

```
    <security:authentication-manager>
        <security:authentication-provider user-service-ref="inMemoryUserServiceWithCustomUser"/>
    </security:authentication-manager>
    <bean id="expressionHandler"
              class="com.apress.pss.terrormovies.security.
CustomWebSecurityExpressionHandler"/>
    <bean id="inMemoryUserServiceWithCustomUser"
              class="com.apress.pss.terrormovies.spring.
CustomInMemoryUserDetailsManager">
        <constructor-arg>
        <list>
            <bean class="com.apress.pss.terrormovies.model.User">
                <constructor-arg value="admin"/>
                <constructor-arg value="admin"/>
                <constructor-arg>
                    <list>
                        <bean class="org.springframework.security.core.authority.
                         SimpleGrantedAuthority">
                         <constructor-arg value="ROLE_ADMIN"/>
                        </bean>
                    </list>
                </constructor-arg>
                <constructor-arg value="Scarioni"/>
                <constructor-arg value="19"/>
            </bean>
        </list>
        </constructor-arg>
    </bean>

    <bean id="logoutRedirectToAny"
class="org.springframework.security.web.authentication.logout.
SimpleUrlLogoutSuccessHandler">
        <property name="targetUrlParameter" value="redirectTo"/>
    </bean>
    <bean id="serverErrorHandler"
class="com.apress.pss.terrormovies.security.ServerErrorFailureHandler"/>
</beans>
```

7. Restart the application, go to http://127.0.0.1:8080/admin/movies, and log in with **admin**/**admin**. You should be able to reach the page without problems.

8. Change the age of the user in the `applicationContext-security.xml` file to 17, restart the application, and go to http://127.0.0.1/admin/movies. Log in with **admin/admin.** You should be denied access to the page. Figure 4-26 shows this last case.



Figure 4-26. *Access denied because the user is underage*

As you can see from the preceding description, SpEL can be really powerful and you can make it do almost anything you want, provided that you execute the expressions in the context of the object you need. Basically, you could extend the expression root as much as you want to handle different methods that you can then use in your expressions.

Switching to a Different User

Sometimes a user (normally an admin user) needs to execute an operation with the permissions of a different user. Suppose that in our dumb application we have a new URL and access to it is allowed only to users with `role ROLE_USER`. This role allows us to retrieve information for users of the application. Let's create the example:

1. Create a new controller named `MovieController` in the package `com.apress.pss.terrormovies.controller` that looks like Listing 4-25.

   Listing 4-25. MovieController

```
package com.apress.pss.terrormovies.controller;

import java.util.HashMap;
import java.util.Map;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
@RequestMapping("/movies")
public class MovieController {

    private Map<Integer, String[]> likedMovies;

    public MovieController(){
```

```
            likedMovies = new HashMap<Integer, String[]>();
            likedMovies.put(1, new String[]{"Die Hard", "Lethal Weapon"});
        }
        @RequestMapping(method = RequestMethod.GET, value = "/member/{id}")
        @ResponseBody
        public String getMoviesForMember(@PathVariable int id) {
            StringBuilder builder = new StringBuilder();
            for(String movie:likedMovies.get(id)){
                builder.append(movie);
            }
            return builder.toString();
        }
    }
```

2. Add the following `<intercept-url>` element as a child of the `<http>` element in the file `applicationContext-security.xml`:

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

3. Create a new user in the configuration file, making the bean definition for the bean with the ID `inMemoryUserServiceWithCustomUser` look like Listing 4-26.

Listing 4-26. The inMemoryUserServiceWithCustomUser bean with a new standard not admin user

```
<bean id="inMemoryUserServiceWithCustomUser"
          class="com.apress.pss.terrormovies.spring.CustomInMemoryUserDetailsManager">
        <constructor-arg>
        <list>
            <bean class="com.apress.pss.terrormovies.model.User">
                <constructor-arg value="admin"/>
                <constructor-arg value="admin"/>
                <constructor-arg>
                    <list>
                        <bean class="org.springframework.security.core.authority.
                        SimpleGrantedAuthority">
                          <constructor-arg value="ROLE_ADMIN"/>
                        </bean>
                    </list>
                </constructor-arg>
                <constructor-arg value="Scarioni"/>
                <constructor-arg value="18"/>
            </bean>
            <bean class="com.apress.pss.terrormovies.model.User">
                <constructor-arg value="paco"/>
                <constructor-arg value="tous"/>
                <constructor-arg>
                    <list>
                        <bean class="org.springframework.security.core.authority.
                        SimpleGrantedAuthority">
                          <constructor-arg value="ROLE_USER"/>
                        </bean>
```

```
                    </list>
                </constructor-arg>
                <constructor-arg value="Miranda"/>
                <constructor-arg value="20"/>
            </bean>
        </list>
        </constructor-arg>
    </bean>
```

4. Create the Movie model class from Listing 4-27.

Listing 4-27. Movie model class

```
package com.apress.pss.terrormovies.model;

public class Movie {
    private String name;
    private String budget;

    public Movie(String name, String budget) {
        super();
        this.name = name;
        this.budget = budget;
    }

    public String getName() {
        return name;
    }

    public String getBudget() {
        return budget;
    }

}
```

5. Restart the application, go to the URL http://127.0.0.1:8080/movies/member/1. Log in with username **paco** and the password **tous**. You should get the movie list on the page. Figure 4-27 shows this page.

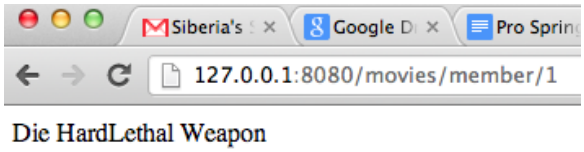


Figure 4-27. *A user with access to the movie list through the role ROLE_USER*

Download from Wow! eBook <www.wowebook.com>

6. Log out by visiting the URL `/j_spring_security_logout`. Visit the URL `http://127.0.0.1:8080/movies/member/1`. Log in again with username **admin** and the password **admin**. You should be denied access to the page.

7. Now let's add switch user functionality. Switch user functionality works using two predefined URLs in the application, which are `/j_spring_security_switch_user` to switch to the target user, and `/j_spring_security_exit_user` to return back from the switched user to the original one. Both URLs need to be secured so that only the users who are allowed to switch users can use them. In this case, they will be secured with a ROLE_ADMIN attribute. Add the following two lines as children of the `<http>` element in the configuration file:

```
<security:intercept-url pattern="/j_spring_security_switch_user"  access=
"hasRole('ROLE_ADMIN')" />
<security:intercept-url pattern="/j_spring_security_exit_user"  access=
"hasRole('ROLE_ADMIN')" />
```

8. Add the filter into the filter chain, because it is not configured by default. Add the following as children of the `<http>` element:

```
<security:custom-filter ref="switchUser" before="FILTER_SECURITY_INTERCEPTOR"/>
```

9. Create the following bean somewhere in the applicationContext-security.xml file:

```
<bean id="switchUser"   class="org.springframework.security.web.authentication.
switchuser.SwitchUserFilter">
     <property name="userDetailsService" ref="inMemoryUserServiceWithCustomUser"/>
     <property name="targetUrl" value="/"/>
</bean>
```

10. Restart the application, visit `http://127.0.0.1:8080/movies/member/1`, and log in with **admin**/**admin**. You will get an "Access Denied" message as you saw before.

11. Now visit the URL `http://127.0.0.1:8080/j_spring_security_switch_user?j_username=paco`. This will take care of switching to the user paco.

12. Now visit the URL `http://127.0.0.1:8080/movies/member/1` again. This time, you can see the movie list!

13. Visit the URL `http://127.0.0.1:8080/j_spring_security_exit_user`.

14. If you visit `http://127.0.0.1:8080/movies/member/1`,, you are denied access again.

The way this whole process works is by creating a new `Authentication` object with the user whom you are trying to impersonate as the principal. This happens inside the `SwitchUserFilter`. Then the filter creates a new Authority for this new Authentication of type `SwitchUserGrantedAuthority`, which contains the original Authentication inside. When you decide to exit the switched user, you can get back to your original Authentication.

Session Management

Another area of Spring Security's web support is the management of user sessions. One very important thing to do regarding sessions is to make sure you create a new session ID when a user authenticates successfully. Doing this reduces the likelihood of session fixation attacks, in which one user sets another user's session identifier to

impersonate him in the application. Spring Security also offers a feature you can use to specify the number of concurrent sessions that the same user can have open at any given time.

These two features come in the form of the two classes `SessionFixationProtectionStrategy` and `ConcurrentSessionControlStrategy` (with the second one being a subclass of the first). Both classes implement `SessionAuthenticationStrategy`. These strategies are invoked from `AbstractAuthenticationProcessingFilter` and the `SessionManagementFilter`. Let's see how they work.

`SessionFixationProtectionStrategy` is already configured by default in the `UsernamePasswordAuthenticationFilter` that is configured in the application. So when you log in, this strategy will be invoked. When the strategy is invoked, it retrieves the current session (which is normally the anonymous session) and invalidates it. Then it immediately creates a new one. It also tries to migrate certain attributes—normally, the ones used by Spring Security itself, but a list can also be specified.

To summarize this strategy, when you log in it invalidates the current session, creates a new one, and copies certain attributes from the old one to the new one.

`ConcurrentSessionControlStrategy` needs to be configured by hand. To do this, you need to add the `<session-management>` element with the following content as a child of the `<http>` element in the configuration file applicationContext-security.xml:

```
                <security:session-management>
        <security:concurrency-control max-sessions="1" />
</security:session-management>
```

By default, this strategy determines that a maximum of one session can be active for any user at any given time. Let's try this out. You will need two different browsers. In the following step-by-step explanation, I assume Chrome and Firefox are accessible to you:

1. Make sure your applicationContext-security.xml looks like Listing 4-28.

   Listing 4-28. applicationContext-security.xml for testing concurrent session management

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd">

    <security:http auto-config="true" use-expressions="true" >
        <security:expression-handler ref="expressionHandler"/>
        <security:intercept-url pattern="/j_spring_security_switch_user"
        access="hasRole('ROLE_ADMIN')" />
        <security:intercept-url pattern="/j_spring_security_exit_user"
        access="hasRole('ROLE_ADMIN')" />
        <security:intercept-url pattern="/admin/*"
            access="(isAnonymous() ? false : principal.lastname == 'Scarioni') and over18" />
        <security:intercept-url pattern="/movies/**/*"  access="hasRole('ROLE_USER')" />

            <security:remember-me key="terror-key" />
            <security:logout delete-cookies="JSESSIONID"
             success-handler-ref="logoutRedirectToAny"/>
            <security:form-login  login-page="/custom_login"
                  authentication-failure-handler-ref="serverErrorHandler"
```

```
                    username-parameter="user_param" password-parameter="pass_param"/>
                <security:custom-filter ref="switchUser"
                before="FILTER_SECURITY_INTERCEPTOR"/>

                <security:session-management>
                        <security:concurrency-control max-sessions="1" />
                </security:session-management>
            </security:http>

            <security:authentication-manager>
                <security:authentication-provider user-service-ref="inMemoryUserServiceWithCustomUser"/>
            </security:authentication-manager>
            <bean id="expressionHandler"

            class="com.apress.pss.terrormovies.security.CustomWebSecurityExpressionHandler"/>
            <bean id="inMemoryUserServiceWithCustomUser"
                      class="com.apress.pss.terrormovies.spring.
        CustomInMemoryUserDetailsManager">
                <constructor-arg>
                <list>
                    <bean class="com.apress.pss.terrormovies.model.User">
                        <constructor-arg value="admin"/>
                        <constructor-arg value="admin"/>
                        <constructor-arg>
                            <list>
                                <bean class="org.springframework.security.core.authority.
                                 SimpleGrantedAuthority">
                                  <constructor-arg value="ROLE_ADMIN"/>
                                </bean>
                            </list>
                        </constructor-arg>
                        <constructor-arg value="Scarioni"/>
                        <constructor-arg value="19"/>
                    </bean>
                    <bean class="com.apress.pss.terrormovies.model.User">
                        <constructor-arg value="paco"/>
                        <constructor-arg value="tous"/>
                        <constructor-arg>
                            <list>
                                <bean class="org.springframework.security.core.authority.
                                 SimpleGrantedAuthority">
                                  <constructor-arg value="ROLE_USER"/>
                                </bean>
                            </list>
                        </constructor-arg>
                        <constructor-arg value="Miranda"/>
                        <constructor-arg value="20"/>
                    </bean>
                </list>
                </constructor-arg>
            </bean>
```

```
    <bean id="logoutRedirectToAny"

class="org.springframework.security.web.authentication.logout.
SimpleUrlLogoutSuccessHandler">
        <property name="targetUrlParameter" value="redirectTo"/>
    </bean>
    <bean id="serverErrorHandler" class="com.apress.pss.terrormovies.security.
     ServerErrorFailureHandler"/>
    <bean id="switchUser" class="org.springframework.security.web.authentication.
     switchuser.SwitchUserFilter">
       <property name="userDetailsService" ref="inMemoryUserServiceWithCustomUser"/>
       <property name="targetUrl" value="/"/>
    </bean>
</beans>
```

2. Restart the application.
3. Open Chrome, visit the URL `http://127.0.0.1:8080/admin/movies`. Log in with a username and password of **admin**. You should be able to access the page without a problem.
4. Open Firefox, and visit the URL `http://127.0.0.1:8080/admin/movies`. Log in with a username and password of **admin**. You should be able to access the page without a problem.
5. Go to Chrome, and refresh the page. You will get the following message: "This session has been expired (possibly due to multiple concurrent logins being attempted as the same user)." This message explicitly indicates what the problem is. Figure 4-28 shows this page.

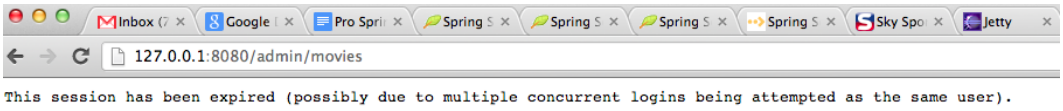


Figure 4-28. *Error related to concurrent sessions*

Now let's allow two sessions at the same time. Change the value in the attribute `max-sessions` of the element `<concurrency-control>` in the configuration file from 1 to 2. Restart the application, and follow the same flow as before. This time, you should have both sessions active at the same time.

Using Different Pattern Matchers for Matching Requests

So far in your `<intercept-url>` elements, you have been using `Ant` expressions for the "pattern" attribute.

On startup, Spring Security creates a `Map<RequestMatcher, Collection<ConfigAttribute>>` when it is parsing the namespace information of the `<intercept-url>` elements. At runtime, when a request arrives in the system, `DefaultFilterInvocationSecurityMetadataSource` sends the request to each of the configured `RequestMatcher` instances (more precisely, to one of its implementations) to see if they match that particular pattern and then retrieves the attribute needed to access the pattern.

Spring Security offers a few `RequestMatcher` implementations beyond the `Ant` one. It basically supports standard Java regular expressions and standard Java case-insensitive regular expressions. To enable support for using these matchers, you simply need to add the attribute `request-matcher` to the `<http>` element specifying the type of matcher your patterns will use within the values `ant`, `regex`, and `ciRegex`. I won't cover this here in any depth, because the idea is the same as for using `Ant` expressions.

Forcing the Request to HTTPS

By default, the Spring Security–enabled application serve all content through the normal HTTP channel. However, you can configure them so that they automatically ensure a particular web request is delivered over the HTTPS channel.

■ **Note** I'm assuming that, in general, you know the advantages of using HTTPS over HTTP. However, I will give you a brief reminder. HTTPS (or Hypertext Transfer Protocol Secure) is a communication protocol that combines the standard HTTP protocol with Secure Sockets Layer (SSL) or TLS for security purposes. HTTPS achieves security in two forms. First, it allows a connecting client to authenticate the web server that it is connecting to, ensuring that it is connecting to the proper certified website. This is done with the use of server-side certificates. The other security concern addressed by HTTPS is the encryption of the information that is exchanged between client and server. HTTPS ensures that the information cannot be read by a third party while it is being exchanged. You can find more extensive information on this topic in Wikipedia (http://en.wikipedia.org/wiki/HTTP_Secure) and in many other places.

Setting this up is straightforward; you simply need to add a new configuration element to your Spring Security configuration. In any `<intercept-url>` element in which you want to configure HTTPS access, you add the attribute `requires-channel="https"` to it. Let's do that with both the login page and the login form's post URLs. You add these two elements as children of the `<http>` element in the configuration file:

```
<security:intercept-url  pattern="/custom_login"  requires-channel="https"/>
<security:intercept-url  pattern="/j_spring_security_check"  requires-channel="https"/>
```

If you do this, however, your current application won't work. The problem is that the Maven Jetty plugin you use to start the application, by default, doesn't recognize the 8443 port for SSL communication. So the only thing you need to do here is replace the current pom.xml's plugin section with the one in Listing 4-29.

Listing 4-29. pom.xml plugin section with Jetty SSL support for HTTPS on port 8443

```
<plugins>
    <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>keytool-maven-plugin</artifactId>
        <executions>
            <execution>
                <phase>generate-resources</phase>
                <id>clean</id>
                <goals>
                    <goal>clean</goal>
                </goals>
            </execution>
            <execution>
                <phase>generate-resources</phase>
                <id>genkey</id>
                <goals>
                    <goal>genkey</goal>
                </goals>
            </execution>
        </executions>
```

```
        <configuration>
            <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
            <dname>cn=apress.pss</dname>
            <keypass>jetty8</keypass>
            <storepass>jetty8</storepass>
            <alias>jetty6</alias>
            <keyalg>RSA</keyalg>
        </configuration>
    </plugin>
    <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>8.1.1.v20120215</version>
        <configuration>
        <connectors>
        <connector implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
          <port>8080</port>
          <maxIdleTime>60000</maxIdleTime>
        </connector>
        <connector implementation="org.eclipse.jetty.server.ssl.SslSocketConnector">
          <port>8443</port>
          <maxIdleTime>60000</maxIdleTime>
          <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
          <password>jetty8</password>
          <keyPassword>jetty8</keyPassword>
        </connector>
        </connectors>
      </configuration>
    </plugin>
</plugins>
```

After that last step, you can restart the application and visit the URL `http://localhost:8080/admin/movies` as you have always done in this chapter. This time, the application will automatically redirect to the URL `https://localhost:8443/custom_login`. (The browser will probably show the typical "insecure certificate" warning, which you can see in Figure 4-29, but you can safely proceed by clicking the "Proceed anyway" option, or the similar option in other browsers.)

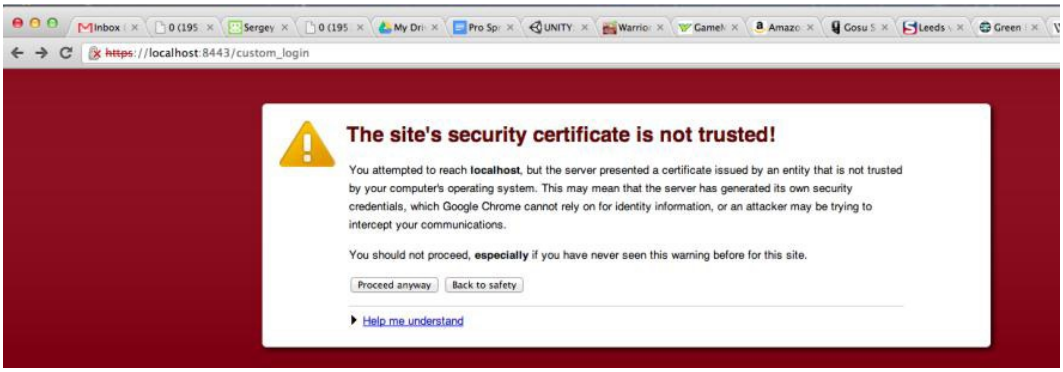


Figure 4-29. *Application with HTTPS channel security and a self-signed certificate*

The channel functionality works in the following way.

When the application starts up, the namespace parsing mechanism will note the presence of the attribute `requires-channel` in the `<intercept-url>` elements. The class `HttpConfigurationBuilder` finds this attribute and then creates a map of `RequestMatcher`, `ChannelAttributeFactory`. `ChannelAttributeFactory` creates `ConfigAttribute` instances based on the value of the `requires-channel` attribute. For example, if the required channel is HTTPS, it creates an instance of an implementation of `ConfigAttribute` with the value REQUIRES_SECURE_CHANNEL associated with the appropriate URL pattern. After the parser finds at least one `intercept-url` with the `requires-channel` attribute configured, it instantiates a new `ChannelProcessingFilter` bean configured with a `ChannelDecisionManager` implementation for deciding which channel to use for each request.

When a request is made, it will, as normal, travel through the filter chain. There is a filter in the filter chain called `ChannelProcessingFilter`, as explained in the previous paragraph. This filter delegates to its configured `ChannelDecisionManager` implementation the responsibility of deciding what to do with the request. The decision manager, with the aid of certain helper classes, decides if the request can proceed or if, on the contrary, the requested channel is not admitted by the requested URL. In the latter case, it delegates to a `ChannelProcessor` and a redirection strategy to send a redirect response to the proper channel URL.

Basically, in this example, this filter looks at the URL `http://localhost:8080/custom_login`, finds that this URL matches an `<intercept-url>` pattern that has the `config` requirement `REQUIRES_SECURE_CHANNEL`, and then calls the `RetryWithHttpsEntryPoint` that, in turn, invokes the `RedirectStrategy` implementation's `sendRedirect` method, which tries to redirect to `https://localhost:8443/custom_login`.

Using the JSP Taglib

Spring Security's web offering comes with a nice suite of view-layer tags you can use to configure the presentation of your application according to the user who is currently logged in. The taglib is oriented mostly to JSPs as a view technology, although in Spring Security 3.1 a new class hierarchy was introduced to support different view-rendering technologies.

The taglib comes packed with plenty of security-oriented tags and attributes. Here is a full list of supported tags and their attributes:

- `authorize` Handles the authorization concerns of the application in the view layer. It determines whether to hide or show data based on whether the requesting user has the proper authorization rights as expressed by the various attributes of this tag.
  - `access` Allows you to use a SpEL expression to determine whether the current user is allowed to see the content inside the tag. This attribute uses the same mechanism that you saw when I was explaining the use of expressions for securing URLs. As a matter of fact, to use these attributes you need to have the attribute `use-expressions="true"` in the `<http>` element in your Spring Security configuration file.
  - `url` A URL within the application. When using this attribute, the `AccessDecisionManager` decides whether the current user has access to the requested URL.
  - `method` To be used in combination with the `url` attribute to determine at a finer-grained level whether or not the content inside the tag should be shown. The decision is based on which HTTP method was used in the current request.
  - `var` A page-scoped variable in which the result of the tag evaluation will be written so that it can be used later instead of making the condition evaluation again.
  - `ifNotGranted` Supports a comma-separated list of role names. It will show the content of the tag only if the user doesn't have any of these roles.
  - `ifAllGranted` Supports a comma-separated list of role names. It will show the content of the tag only if the user has all of these roles.

    - `ifAnyGranted` Supports a comma-separated list of role names. It will show the content of the tag only if the user has any of these roles.
- `authentication` This tag give you access to the `Authentication` object corresponding to the logged-in user.
    - `property` A property of the Authentication object that should be output, allowing the use of nested properties.
    - `var` The name of the property to which the value of the property from the Authentication will be assigned.
    - `htmlEscape` A Boolean that indicates if the tag needs HTML escaping.
    - `scope` The scope in which the `var` variable is defined.
- `accesscontrollist` This tag allows you to carry the ACL functionality of Spring Security to the view layer. It also allows you to render content conditionally based on the rights of the user on domain objects.
    - `hasPermission` A comma-separated list of permissions to be evaluated against a domain object.
    - `domainObject` The domain object against which the permissions are evaluated.
    - `var` A page-scoped variable that will hold the result of the evaluation of the tag so that it doesn't have to be evaluated again in the future if the same condition is needed in the page.

I will cover the first two tags here; for the third, please refer to Chapter 7.

Let's see this at work. First, let's create a new method in `MovieController` that allows both `ROLE_USER and ROLE_VIP` access, and then let's work in a new JSP with different configurations:

1. Create the method from Listing 4-30 in the `MovieController` class. This new method simply creates one list of movies and passes it to the view as a `ModelAndView` Spring MVC object.

    Listing 4-30. getAllMovies returns a list of movies model and a view "movies" to be rendered

    ```
    @RequestMapping(method = RequestMethod.GET, value = "/")
        public ModelAndView getAllMovies() {
            ModelAndView mv = new ModelAndView("movies");
            List<Movie> movies = new ArrayList<>();
            movies.add(new Movie("Die hard", "25000000"));
            movies.add(new Movie("Lethatl Weapon", "30000000"));
            mv.addObject("movies", movies);
            return mv;
        }
    ```

2. Next create a `movies.jsp` file in the `WEB-INF/views` folder with the content from Listing 4-31.

    Listing 4-31. movies.jsp uses Spring Security tags to filter and output content

    ```
    <%@ page language="java" contentType="text/html; charset=ISO-8859-1"
        pageEncoding="ISO-8859-1"%>
    <%@ taglib prefix="security"
        uri="http://www.springframework.org/security/tags"%>
    ```

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Terror movies</title>
</head>
<body>
    Mr  <security:authentication property="principal.lastname"/>
    <br/><br/>
    <c:forEach items="${movies}" var="movie">
        <security:authorize access="hasRole('ROLE_USER')">
            ${movie.name}<br />
        </security:authorize>
        <security:authorize access="hasRole('ROLE_VIP')">
            ${movie.budget}<br />
        </security:authorize>
    </c:forEach>
</body>
</html>
```

3. Create a new `intercept-url` for this new content, and create a new user with the roles `ROLE_VIP` and `ROLE_USER`. You do all this by adding `<security:intercept-url pattern="/movies/*" access="hasAnyRole('ROLE_USER','ROLE_VIP')"/>` in the configuration file applicationContext-security.xml as a child of the `<http>` element and adding a new user to the `inMemoryUserServiceWithCustomUser` bean's constructor list with the roles `ROLE_VIP,ROLE_USER`:

```
<bean class="com.apress.pss.terrormovies.model.User">
            <constructor-arg value="lucas"/>
            <constructor-arg value="fernandez"/>
            <constructor-arg>
                <list>
                    <bean
class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                          <constructor-arg value="ROLE_VIP"/>
                    </bean>
                    <bean class="org.springframework.security.core.authority.
                    SimpleGrantedAuthority">
                      <constructor-arg value="ROLE_USER"/>
                    </bean>
                </list>
            </constructor-arg>
            <constructor-arg value="Silva"/>
            <constructor-arg value="20"/>
</bean>
```

The code from the previous listing is very straightforward. The important part is that you are using the `standard jstl` tag library to iterate through the list of movies you are expecting to get back from your controller in the model. Then, you use the `<authorize>` security tag that I explained before to secure certain parts of the view. You can see that users with the role ROLE_USER can see the movie name. Users with the role ROLE_VIP can see the movie budget.

4. Restart the application now, visit the URL http://localhost:8080/movies/, and log in with a username of **paco** and a password of **tous**. You should see the following text:

   Mr Miranda
   Die hard
   Lethal Weapon

5. Log out by visiting the URL http://localhost:8080/j_spring_security_logout. Then visit the URL http://localhost:8080/movies/ again, and log in with the username **lucas** and the password **fernandez**. You should see the following content:

   Mr Silva
   Die hard
   25000000
   Lethal Weapon
   30000000

You can see how you are using both "authentication" and "authorize" tags in the jsp and how they work. In the "authentication" one, you are accessing the principal and the last name of the user. In the "authorize" one, you are allowing only certain content to be shown based on the role of the logged-in user.

If you want to use the same conditions again later in the page, you can use the `var` attribute to cache the result of the condition evaluation so that you don't need to evaluate it again. For example, replace your `movies.jsp` with the content from Listing 4-32. Now if you log in with a username of **lucas**, a password of **fernandez** and visit the URL http://localhost:8080/movies/, you will see the text "You are a very appreciated VIP user" on the page. If you do the same when logging in with the username of **paco** and a password of **tous**, you won't see the message output on the page.

Listing 4-32. movies.jsp using the var attribute in an authorize tag to cache condition evaluation

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Terror movies</title>
</head>
<body>
    Mr  <security:authentication property="principal.lastname"/>
    <br/><br/>
    <c:forEach items="${movies}" var="movie">
        <security:authorize access="hasRole('ROLE_USER')" var="isUser">
            ${movie.name}<br />
        </security:authorize>
        <security:authorize access="hasRole('ROLE_VIP')" var="isVip">
            ${movie.budget}<br />
        </security:authorize>
    </c:forEach>
    <c:if test="${isVip}">
        You are a very appreciated VIP user
    </c:if>
</body>
</html>
```

Download from Wow! eBook <www.wowebook.com>

Role Hierarchies

So far, you have worked with simple roles assigned to users. For example, a user can have ROLE_USER, another user can have ROLE_ADMIN, and yet another user can have both ROLE_USER and ROLE_ADMIN. However, in reality, the most common thing is that any user with the role ROLE_ADMIN should be allowed access to anything that a user with the role ROLE_USER has access to. There shouldn't really be a need for that admin user to have both roles in order to do the ROLE_USER activities, because ROLE_ADMIN can be seen as a superset of access permissions over that of ROLE_USER.

The logical thing here would be to assign the role ROLE_ADMIN to that user and then create some kind of hierarchy to specify the "contains" relationship of ROLE_ADMIN over ROLE_USER.

Spring allows you to do exactly that with the use of hierarchical roles. You establish a role hierarchy in Spring Security using the greater than symbol (>) in the following way: ROLE_ADMIN > ROLE_USER. You can't chain a hierarchy like this: ROLE_ADMIN > ROLE_USER > ROLE_GUEST. However, you can break that into two individual hierarchy chunks to achieve the same effect: ROLE_ADMIN > ROLE_USER, ROLE_USER > ROLE_GUEST.

Setting hierarchical roles in Spring Security is simple. You need to set up just two classes. The first class is `RoleHierarchyImpl`, which implements `RoleHierarchy`, which defines a single method `public Collection<? extends GrantedAuthority> getReachableGrantedAuthorities(Collection<? extends GrantedAuthority> authorities)`. That method returns all the reachable authorities for a particular list of authorities, which basically means all direct assigned authorities plus all the authorities that are children of those direct authorities. The class `RoleHierarchyImpl` first needs to parse the string of role hierarchies that is passed to it (the ones of the form ROL_ADMIN > ROLE_USER). Then it needs to make those available as `GrantedAuthority` objects to be queried and returned by the class's main method.

The second class to set up is `RoleHierarchyVoter`, which extends from `RoleVoter` and simply delegates to `RoleHierarchyImpl` to retrieve the reachable roles for a particular user. It then uses those roles for voting on accessing or denying access according to the logic from the `RoleVoter` class.

Let's configure this and see how it looks. To make it simple and focus just on this functionality, go ahead and replace your whole applicationContext-security.xml file with the one from Listing 4-33.

Listing 4-33. applicationContext-security.xml for hierarchical roles

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true"
        access-decision-manager-ref="accessDecisionManager">
        <security:intercept-url pattern="/admin/*"
            access="ROLE_ADMIN" />
        <security:intercept-url pattern="/movies/*"
            access="ROLE_USER" />
        <security:intercept-url pattern="/movies/*"
            access="ROLE_GUEST" />
    </security:http>
```

```
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user name="car" password="scarvarez"
                    authorities="ROLE_USER" />
                <security:user name="mon" password="scarvarez"
                    authorities="ROLE_ADMIN" />
                <security:user name="bea" password="scarvarez"
                    authorities="ROLE_GUEST" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>

    <bean id="accessDecisionManager"
        class="org.springframework.security.access.vote.AffirmativeBased">
        <constructor-arg>
            <list>
                <ref bean="roleVoter" />
            </list>
        </constructor-arg>
    </bean>

    <bean id="roleVoter"
        class="org.springframework.security.access.vote.RoleHierarchyVoter">
        <constructor-arg>
            <bean
                class="org.springframework.security.access.hierarchicalroles.RoleHierarchyImpl">
                <property name="hierarchy">
                    <value>
                        ROLE_ADMIN > ROLE_USER
                        ROLE_USER > ROLE_GUEST
                    </value>
                </property>
            </bean>
        </constructor-arg>
    </bean>
</beans>
```

The preceding code demonstrates practically everything I've explained so far in this section. The new things that you haven't seen before are the last bean and its internal anonymous bean. Here you are defining the new `AccessDecisionVoter` that is aware of role hierarchies, and `RoleHierarchy` itself. You are defining a relationship between roles, such as ROLE_ADMIN > ROLE_USER > ROLE_GUEST. Then you are assigning this voter to the `AccessDecisionManager` you configured (the `AffirmativeBased` one).

Let's test this out. Restart the application, and visit the URL `http://localhost:8080/movies/member/1`. As you can see in the configuration file applicationContext-security.xml, this is a URL secured for users with the role ROLE_USER. When the page shows the login form, log in as an admin user. The one you have is "car", so log in with a username of **car** and a password of **scarvarez**. In previous examples, you would not be able to access the page; however, thanks to the hierarchical roles, now you are able to access the page, as Figure 4-30 shows.

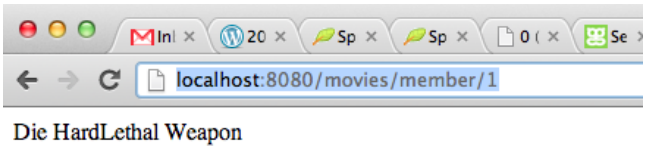


Figure 4-30. *Access is granted using a hierarchical role*

Summary

In this chapter, I covered one of the biggest concerns of the framework: web support in Spring Security. You saw that the main functionality comes in the form of Servlet filters. This is a good thing from a standards point of view, because it means you can leverage Spring Security web support in other frameworks that use the standard Java servlet model. It can be used for simple Servlet applications as well, as shown in Chapter 2.

You should now know a lot of details about the main filters that build the framework, how they work internally, and how they fit within each other and with the rest of the framework. I explained all this in a practical way, trying to solve real-life scenarios (although in a not very real use-case). I also explained how to attack them in a step-by-step process. You should be able to understand both role-based security and the power of using SpEL for security rules. You learned also how to extend the standard SpEL functionality to support new expressions.

You can now use the taglib support included in the framework to customize the user interface, depending on security constraints.

In the next chapter, I will cover the second major concern of Spring Security—namely, method-level security. I will show you how it compares to web-level security, and you will see that you can leverage a lot of your current knowledge to apply it to the method-level security layer.

CHAPTER 5

Securing the Service Layer

This chapter will drill down further into the core functionality of Spring Security.

Unlike in the previous chapter where I focused only on the web-level access to the application, here you will see another important area in which Spring Security can help you secure your applications. It can be seen as a more invasive type of configuration because it involves securing at the code level, while the web layer simply was concerned with URL matching. However, you will see how the elegant way in which Spring can manage aspect-oriented programming (AOP) concerns will make business-level security as unobtrusive as its web-level counterpart.

The Limitations of Web-Level Security

In the previous chapter, you established security constraints at the web layer of the application, mainly at the URL level and the view-rendering layer. This is very nice and powerful; however, it is not a 100 effective solution in every use case. The main concerns with applying only this kind of security are both functional and convenience related, as I will explain next

First, by definition, web-layer security applies only to web applications, which makes it unusable for any other kind of Java application. Although Spring Security's main focus is on securing web applications, there is no reason why some of its parts can't be used for different kinds of applications.

Second, the URL pattern-matching mechanism for security, although flexible, requires the developer to adopt certain rules or conventions just for the sake of security (like creating an /admin/ URL namespace for admin users and setting administrator rules on those URLs).

Third, securing at the URL level creates only a coarse-grained security, as a URL is the entry point into the application. This means that security constraints are enforced on a per-request basis, greatly reducing flexibility. For example, if you want to ensure that a particular Data Access Object (DAO) in your application is called only by an administrator user, you can't do that with web security alone. You need to make sure that all the URLs that call this DAO are secured for Admin users. If, for some reason, you have a URL that is not secured correctly, that request will freely reach the DAO layer, where it could execute a potentially delicate operation.

You can surely find a few more reasons why web-only security is not enough sometimes, or simply not convenient enough to use by itself in your applications.

What Is Business Service-Level Security?

Service-layer security (or more accurately *method-level security*) is a feature of Spring Security you use to enforce security constraints at the method level, much as web-level security does at the URL level.

At its core, method-level security relies on Spring AOP's powerful support for providing you with its services. This is the main difference in implementation with web-based security that depends on Servlet Filters, although it is worth noticing that under the hood most of the core code that will take care of the security constraints is the same. This is of great importance, as it shows good care in designing a set of reusable and encapsulated components in the architecture.

As I said, service-layer security is normally used in combination with web-based security, and I will cover this scenario mostly in this chapter. However, as you will see later, you can use service-layer security by itself without the context of a web application.

The traditional scenario for working with Spring Security is this: You have a web-based application, with a relatively thin web layer, backed by one Spring-implemented business service layer. With regard to security, the web layer is configured to take care of ensuring that there is a user authenticated in the system (that is, it takes care of the authentication part of the security, using forms, http status codes, and so on). The service layer has the authorization rules in per-operation criteria and with the needed level of granularity. Most of the time, you secure the business services; some other times, you might need to secure the DAOs.

Setting Up the Example for the Chapter

Let's start doing some work and see how this whole thing works. You will be using the same application from last chapter, so make sure you have it at hand.

You will use Spring AOP in this chapter. So you need to add support for it in your application. The only thing you will do to support it at the moment is break your classes (the ones you want to decorate with security concerns) into interface-class hierarchies. For a start, you do this with the `AdminController`, creating an `AdminController` interface and an `AdminControllerImpl` implementation class, as you can see in Listing 5-1. You can see that I copied the Spring model view controller (MVC) annotations into this new interface. If I had not done this and had left them on the implementation only, Spring MVC wouldn't be able to find them when looking for a method to handle the incoming requests.

■ **Note** Normally, in simple controllers, you can keep the `@RequestMapping` annotations on the implementing class without needing to create an interface. However, in the case of this example, you need to put the annotation on the interface because you will use security annotations in the class, which automatically will create a proxy object that needs the presence of an interface—for using standard Java Development Kit (JDK) proxy objects, as you will see in the upcoming paragraphs. This proxy won't know about the `@RequestMapping` annotation, so Spring's MVC mechanism won't be able to find the handler methods. Later I will put the security annotations where they belong, in the service layer. I'm putting them now in the controller simply to illustrate the simplest scenario. In real life you would not do it this way and instead you would add it to the service layer as I will show you later in the chapter.

It is important anyway to bear in mind this behavior when working with Spring MVC or any other part of Spring that uses proxy objects around your objects..

Spring AOP comes in two flavors: standard java proxies that work with interfaces, and CGLIB support that creates proxies at the implementation-class level. Each has its strengths and weaknesses. They work in the following way.

- **Using standard JDK proxies** If you have a Spring bean that implements an interface, and it should be enhanced with some AOP concern (for example, if it has configured Spring support of standard AOP concerns like transactionality, security, or synchronicity), Spring creates, when starting up and instantiating the beans, a new class that implements the same interfaces as your original bean, adds the specific functionality that it needs (transaction awareness, security, and so on) to that new class, and then wraps your original object in this new object, creating in essence a "decorated" version of your object. This new decorated object will be the actual bean used in the framework, transparently replacing the original object whenever it is accessed within the Spring application. Basically, if you inject your defined bean into another bean, the object instance that gets injected is the proxy.

- **Using CGLIB** CGLIB, on the other hand, allows you to proxy objects without needing to create an interface for them. CGLIB works by subclassing the classes you want to proxy and adding in these subclasses the needed behavior. It then delegates to the original class of the object for the core behavior.

In general, Spring favors standard interface-based proxies instead of CGLIB, and that is what I will use in this book. Listing 5-1 demonstrates how a new interface is added to allow Spring to use JDK proxies to enable cross-cutting concerns.

Listing 5-1. New AdminController Hierarchy

```
package com.apress.pss.terrormovies.controller;

import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

@RequestMapping("/admin")
public interface AdminController {

@RequestMapping(method = RequestMethod.POST, value = "/movies")
@ResponseBody
public abstract String createMovie(@RequestBody String movie);

@RequestMapping(method = RequestMethod.GET, value = "/movies")
@ResponseBody
public abstract String createMovie();

}

package com.apress.pss.terrormovies.controller;

import org.springframework.security.access.annotation.Secured;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

import com.apress.pss.terrormovies.model.User;

@Controller
@RequestMapping("/admin")
public class AdminControllerImpl implements AdminController{

    @RequestMapping(method = RequestMethod.POST, value = "/movies")
    @ResponseBody
    @Secured("ROLE_ADMIN")
```

```
    public String createMovie(@RequestBody String movie) {
        System.out.println("Adding movie!! "+movie);
        return "created";
    }

    @RequestMapping(method = RequestMethod.GET, value = "/movies")
    @ResponseBody
    @Secured("ROLE_ADMIN")
    public String createMovie() {
        User user = (User)SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        System.out.println("returned movie!");
        return "User "+user.getLastname()+" is accessing movie x";
    }
}
```

The next thing you'll do is activate method-level security in the framework. To do this, you open the applicationContext-security.xml file and leave it as it is shown in Listing 5-2. The line `<security:global-method-security secured-annotations="enabled" />` enables the use of the `@Secured` annotation in the application. As you will see later, you can use other types of annotations as well, but for now let's focus on `@Secured`.

Listing 5-2. applicationContext-security.xml Simplified, with Method Security Configured

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:security="http://www.springframework.org/schema/security"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">

   <security:global-method-security secured-annotations="enabled" />
   <security:http auto-config="true" />
   <security:authentication-manager>
      <security:authentication-provider user-service-ref="inMemoryUserServiceWithCustomUser" />
   </security:authentication-manager>
   <bean id="expressionHandler"
      class="com.apress.pss.terrormovies.security.CustomWebSecurityExpressionHandler" />
   <bean id="inMemoryUserServiceWithCustomUser"
      class="com.apress.pss.terrormovies.spring.CustomInMemoryUserDetailsManager">
      <constructor-arg>
         <list>
            <bean class="com.apress.pss.terrormovies.model.User">
               <constructor-arg value="admin" />
               <constructor-arg value="admin" />
               <constructor-arg>
                  <list>
                     <bean
class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                        <constructor-arg value="ROLE_ADMIN" />
                     </bean>
                  </list>
               </constructor-arg>
```

```
            <constructor-arg value="Scarioni" />
            <constructor-arg value="19" />
         </bean>
         <bean class="com.apress.pss.terrormovies.model.User">
            <constructor-arg value="paco" />
            <constructor-arg value="tous" />
            <constructor-arg>
               <list>
                  <bean
class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                     <constructor-arg value="ROLE_USER" />
                  </bean>
               </list>
            </constructor-arg>
            <constructor-arg value="Miranda" />
            <constructor-arg value="20" />
         </bean>
         <bean
class="com.apress.pss.terrormovies.model.User">
            <constructor-arg value="lucas" />
            <constructor-arg value="fernandez" />
            <constructor-arg>
               <list>
                  <bean
class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                     <constructor-arg value="ROLE_VIP" />
                  </bean>
                  <bean
class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                     <constructor-arg value="ROLE_USER" />
                  </bean>
               </list>
            </constructor-arg>
            <constructor-arg value="Silva" />
            <constructor-arg value="20" />
         </bean>
      </list>
   </constructor-arg>
</bean>
</beans>
```

You can see that I removed a lot from the file, including the `<intercept-url>` elements. That means you are no longer enforcing security at the URL level.

In Listing 5-1, you can see a new annotation: the `@Secured` annotation. You still want your administration operations to be available only to administrator users, so I configured the `AdminController` to be accessible only by users with ROLE_ADMIN in their list of roles. To do that, you simply need to add the annotation `@Secured("ROLE_ADMIN")` to the methods of the class `AdminControllerImpl`.

■ **Note** The `@Secured` annotation also can be used at the class level instead of at the method level. If it's used at the class level, all public methods of the class will inherit the behavior specified by the annotation.

A combination of class-level and method-level annotations also can be used. In this case, method annotations will override the values of class-level annotations.

If you restart the application now and access the URL `http://localhost:8080/admin/movies`, you will be able to access it. It will throw an exception, but one that is related to a class cast exception, as you can see in Figure 5-1. This exception is thrown because the `@Secured` annotation is not being detected by the framework for reasons I will explain next; and there is no `User` object instantiated at the point where the exception is thrown. Instead, the application is using the anonymous authentication, whose principal is a string, which it then fails to cast to a fully built `User` object.

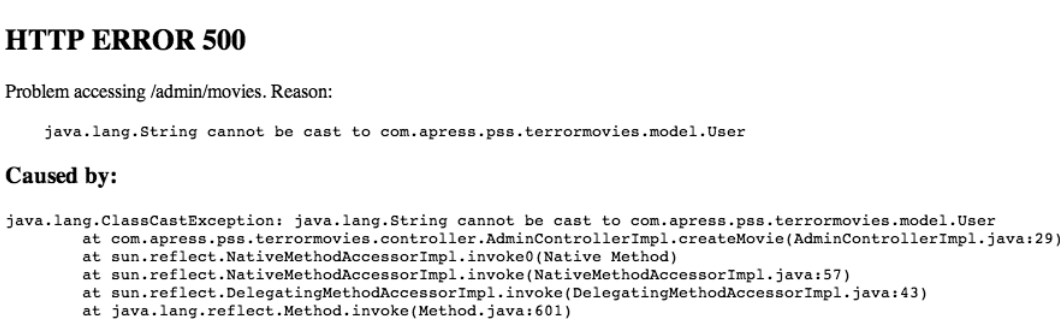


Figure 5-1. *Non-security-related exception trying to access the application because the @Secured annotation is not being picked up*

As I said, you shouldn't have been allowed access to that page, as you have configured the method-level annotation to ensure that only `administrators` can access it. So what is the problem?

The problem is that the XML configuration element `<security:global-method-security secured-annotations="enabled" />` applies only to beans defined in the same file where this element exists. So you need to add it to the terrormovies-servlet.xml file to be able to use the annotations in the controller layer.

■ **Note** The way `global-method-security` works at startup is explained in Chapter 3. Refer to that chapter for an explanation of the startup process.

After you add the element `<security: global-method-security secured-annotations="enabled" />` to terrormovies-servlet.xml, add the corresponding entries in the namespace definitions in the `XML <beans>` element (the ones corresponding to the security namespace that look like `xmlns:security=http://www.springframework.org/schema/security`, which you can copy and paste them from the applicationContext-security.xml file). Then restart the application. You can see that if you try to access the URL `http://localhost:8080/admin/movies`, you get redirected to the already familiar login form. This means that the security annotation is now finally working. You can see this in Figure 5-2.

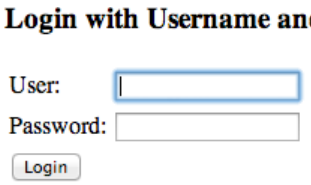


Figure 5-2. *Login form shown after ensuring that the @Secured annotation is being picked up*

How the Described Actions Happen Under the Hood

The main work for enforcing the authorization process now happens in the `MethodSecurityInterceptor`, which is the method-level equivalent of the `FilterSecurityInterceptor`. (They both extend from the same abstract class `AbstractSecurityInterceptor`.) Remember that the main difference is that, in this case, Spring AOP mechanisms get into action to enforce security by decorating the method call, while in the filter case, the Servlet Filter standard itself helps to validate and enforce the constraints by intercepting the HTTP request.

Figure 5-3 shows how the `MethodSecurityInterceptor` executes within a proxy object around the target object (`AdminControllerImpl`).

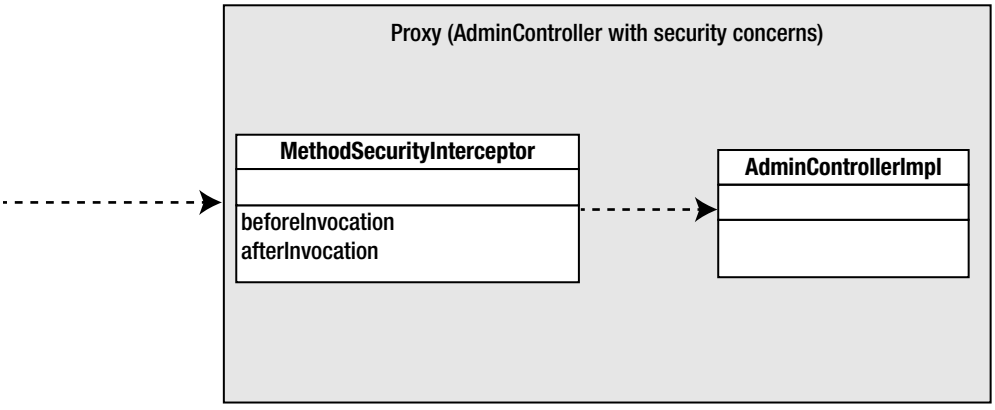


Figure 5-3. *AdminController proxied with security*

The whole process works like this: As I explained before, on startup our `AdminControllerImpl` bean gets wrapped by Spring AOP into a Spring Security–aware `AdminController` implementation, also known as the `Proxy` object. This proxy has all the information it needs to enforce security at the method level. The main part of this information is an instance of `MethodSecurityInterceptor` that lives inside the `Proxy` object.

The `Proxy` object takes the target object (the original bean instance of `AdminControllerImpl`), the invoked method, and the arguments to the method (if there is any) and sends them to an instance of `org.springframework.aop.framework.JdkDynamicAopProxy`. This object sees that the `MethodSecurityInterceptor` is in the list of advisors—instances of classes implementing the `org.springframework.aop.Advisor` interface that contain advice (`http://en.wikipedia.org/wiki/Advice_in_aspect-oriented_programming`)—of the bean and creates an `org.springframework.aop.framework.ReflectiveMethodInvocation` object. This object's `proceed` method is invoked, which in turn invokes the `MethodSecurityInterceptor`'s `invoke` method, passing itself as a parameter. Figure 5-4 shows graphically this relatively complex setup.

Download from Wow! eBook <www.wowebook.com>

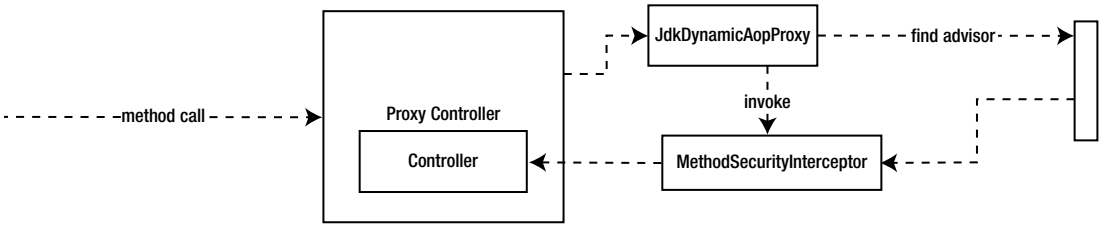


Figure 5-4. *JdkDynamicAopProxy is called by the proxy and invokes the interceptor*

So the `MethodSecurityInterceptor` will do this before invocation; after invocation, it checks around the real target method logic, the same way that the `FilterSecurityInterceptor` did.

The `MethodSecurityInterceptor`'s before-and-after logic is exactly the same as that for `FilterSecurityInterceptor`. The difference is the implementation strategy of `SecurityMetadataSource` that each approach uses. `MethodSecurityInterceptor` uses an instance of `DelegatingMethodSecurityMetadataSource`, which knows how to extract security metadata out of a `ReflectiveMethodInvocation` instance. In the current execution, this `SecurityMetadataSource` finds the value `"[ROLE_ADMIN]"` in the method invocation. Later, this will be used by the `AccessDecisionManager`'s current implementation (normally, `ProviderManager`) and the `RoleVoter` to decide whether or not access is granted, in exactly the same way as with the `FilterSecurityInterceptor`.

Creating a Business Layer in Your Application

Although I have shown you how to use security annotations at the method level for your controllers, the truth is that you can use these annotations in any layer of your application, and normally they are used at the business service layer. As a matter of fact, you probably won't ever use them in the controller layer. I only used them in that layer in the first part of the example to introduce the concepts step by step.

The only constraints to using security annotations in the service layer, as explained before, are that they have to be used in Spring managed beans, which can be proxied, and they can be applied only in public methods that can be proxied.

■ **Note** Remember that Spring creates a proxy that wraps your objects. This proxy is visible only to external callers of methods in the object through the proxy interface. Once a call reaches the object, any internal method calls within the object itself are not proxied and go directly against the object itself, so no AOP is applied in internal calls.

For continuing with the explanations for this chapter, let's create a thin business layer where you will start adding your security constraints.

In a new package named `com.apress.pss.terrormovies.service`, let's create the interface and class from Listing 5-3.

Listing 5-3. MoviesService and MoviesServiceImpl

```
package com.apress.pss.terrormovies.service;
import com.apress.pss.terrormovies.model.Movie;
```

```
public interface MoviesService {
    Movie getMovieByName(String name);
}

package com.apress.pss.terrormovies.service;

import java.util.HashMap;
import java.util.Map;

import org.springframework.security.access.annotation.Secured;

import com.apress.pss.terrormovies.model.Movie;

public class MoviesServiceImpl implements MoviesService{

private static final Map<String, Movie> MOVIES = new HashMap<String, Movie>();

static{
        MOVIES.put("die hard", new Movie("Die Hard", "20000000"));
    }

    @Secured("ROLE_USER")
    public Movie getMovieByName(String name) {
        return MOVIES.get(name.toLowerCase());
    }

}
```

Next, define a bean in the applicationContext-security like this: <bean id="moviesService" class="com.apress.pss.terrormovies.service.MoviesServiceImpl"/>. Then replace our MovieController with the one from Listing 5-4.

Listing 5-4. MovieController functionality simply Delegates to the MoviesService for Getting the Model which in this case is a Movie

```
package com.apress.pss.terrormovies.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.ResponseBody;

import com.apress.pss.terrormovies.model.Movie;
import com.apress.pss.terrormovies.service.MoviesService;

@Controller
@RequestMapping("/movies")
```

```
public class MovieController {

    @Autowired
    private MoviesService moviesService;

    @RequestMapping(method = RequestMethod.GET, value = "/{name}")
    @ResponseBody
    public String getMovieByName(@PathVariable String name){
        Movie movie = moviesService.getMovieByName(name);
        return movie.toString();
    }
}
```

Restart the application. If you visit the URL `http://localhost:8080/movies/die hard`, you once again will be redirected to the login page. Log in with `"lucas" as username and "fernandez" as password` and you should be able to access the requested page as expected. You can see that in Figure 5-5. Make sure you have a proper `toString` implemented in your Movie.java file. Listing 5-5 shows the one that gives the output shown in Figure 5-5.

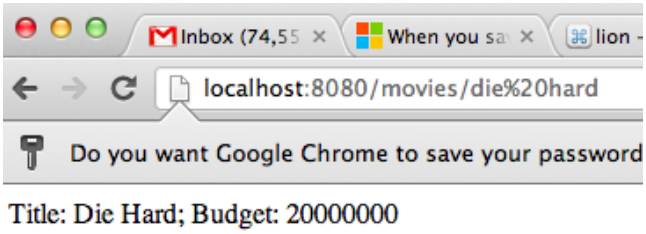


Figure 5-5. *Access granted in the service method for seeing the movie*

Listing 5-5. The toString Method in the Movie.java Class File

```
public String toString(){
   return "Title: "+name+"; Budget: "+budget;
}
```

So you have moved the security authorization logic to the business layer and left the controller with the single responsibility of coordinating interactions and mapping views to models. This should be the practice you implement most often, and as I said before, you normally won't have security configured at the method level in controllers.

Note that although the authorization process has been moved to the business layer, the authentication process still exists in the web layer and the filter-chain logic. This is the most common scenario where Spring Security fits in: a web-based front end application with a Spring-powered back-end business layer. You will see later how you can use Spring Security in a non-web-based application.

@RolesAllowed Annotation

The `@javax.annotation.security.RolesAllowed` annotation serves the same functionality as the Spring-specific `@Secured` annotation. Its only advantage over `@Secured` is that it is a Java standard annotation, developed with the Java Specification jsr250, and its semantics can be carried over to a non-Spring Java Enterprise Edition (EE) application. This advantage also implies that someone coming from a JavaEE background, without Spring knowledge, can understand what is happening when she sees this annotation in place.

■ **Note** My attitude to standards is that using them is good, as long as they provide you with the functionality you need. In my mind, you should not depend on the standards to drive your work. Your work and your needs should be driving the standards, and the minute you can't do it with standards you should consider finding a simpler solution that works (or develop it yourself if it is worth the effort), instead of trying to tweak the standard to work in an awkward way just to say that you still work within the standard.

I used to be a bigger believer in standards in Java, but my attitude has changed a lot through the years I have been developing software. I tend to look for simple solutions that, to my eyes, look more elegant than unjustified complex systems. My last experience was a few years ago when comparing different integration solutions (ESB-like tools). I had no problems at all understanding and working with Mule ESB, Apache Camel, and Spring Integration. But trying to get an understanding of OpenESB and Java Business Integration (JBI) was definitely not as straightforward.

This is a large subject of debate and is definitely outside the scope of this book.

To make `@RolesAllowed` work in your application is pretty simple. You have to do three simple steps:

1. Add the dependency from Listing 5-6 to the pom.xml file.
2. In the element `<security:global-method-security>` in the file applicationContext-security.xml, add the attribute: `jsr250-annotations="enabled"`.
3. In the annotated method `getMovieByName` inside the `MoviesServiceImpl` class, replace the current `@Secured("ROLE_USER")` annotation with the following: `@RolesAllowed("ROLE_USER")`.

Now if you restart the application and access "`http://localhost:8080/movies/die hard`" from your browser, the behavior should be the same as before.

Listing 5-6. jsr250 Maven Dependency for Standard Annotations

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>jsr250-api</artifactId>
    <version>1.0</version>
</dependency>
```

When you're using `@RolesAllowed` instead of `@Secured`, the `org.springframework.security.access.annotation.Jsr250Voter` will be used by the access decision manager to make the authorization decisions. Again, it behaves pretty much like the `RoleVoter`, but it will make a decision based on the jsr250 annotations.

Securing the Application Using SpEL Expressions

As in the case of web-based security, at the method level, you can use SpEL expressions to define security constraints. For using expressions at the method level, you need to add the `pre-post-annotations="enabled"` attribute to the element `<security:global-method-security>` in the applicationContext-security.xml file.

This attribute gives you access to four different annotations in your methods. The following annotations are all in the org.springframework.security.access.prepost package in the core module of the framework:

- **`@PreFilter`** This annotation is used to specify a filtering SpEL expression to be applied to an argument of the invoked method. The argument to which it is applied needs to be a `java.util.Collection` type that responds to the remove method. Basically, the SpEL expression is evaluated against each element in the collection, and if the expression evaluates to false, the element will be removed from the collection before it is passed to the method.
- **`@PreAuthorize`** This annotation is used with a SpEL expression that is evaluated to determine whether or not access to the method is granted.
- **`@PostFilter`** This annotation is used for specifying SpEL expressions that will be evaluated against the returning value of the secured method. The returned value needs to be a `java.util.Collection` implementation that responds to the remove method. The expression is evaluated against each of the elements of the returned collection. If the expression evaluates to false for an element, the element will be removed from the collection.
- **`@PostAuthorize`** In this annotation, a SpEL expression is evaluated after the method returns. You can use this annotation to make access decisions even after the target method has been accessed.

The first two annotations (`@PreFilter` and `@PreAuthorize`) are exercised in the `AbstractSecurityInterceptor`'s `beforeInvocation` method. (Remember that `AbstractSecurityInterceptor` has two concrete implementations: `MethodSecurityInterceptor` and `FilterSecurityInterceptor`. However, the before and after processing are the same in both cases and it is inherited (both pre and post processing functionality) from the abstract class.) They are evaluated by a dedicated `AccessDecisionVoter` implementation. The `org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter` delegates further to an instance of `org.springframework.security.access.expression.method .ExpressionBasedPreInvocationAdvice`, which evaluates the expressions. The two post annotations (`@PostFilter` and `@PostAuthorize`) are used in the `AbstractSecurityInterceptor`'s `afterInvocation` method, which delegates to an instance of `org.springframework.security.access.prepost.PostInvocationAdviceProvider`, which itself delegates to an instance of `org.springframework.security.access.expression.method .ExpressionBasedPostInvocationAdvice` to evaluate the expressions.

Let's take a look at the use of these annotations one by one. Let's start with the `@PreAuthorize` annotation.

In your current `MoviesServiceImpl` class, you replace the `@RolesAllowed("ROLE_USER")` annotation in the method `getMovieByName` with the following one: `@PreAuthorize("#name.length() < 50 and hasRole('ROLE_USER')")`. You then add `import import org.springframework.security.access.prepost.PreAuthorize;` to the list of imports in the class.

Now let's do the following:

1. Restart the application.
2. Access the URL `http://localhost:8080/movies/die hard`. You will be denied access and shown the login form.
3. Fill the login form with the username "lucas" and the password "fernandez". You will get redirected to the correct page—the same one you saw in Figure 5-5.
4. Access the URL `http://localhost:8080/movies/dieharddiedieharddiedieharddiedieharddiedieharddie`. You will be denied access again, getting an "Access is denied" message page as shown in Figure 5-6. The following paragraphs explain what happens under the hood.

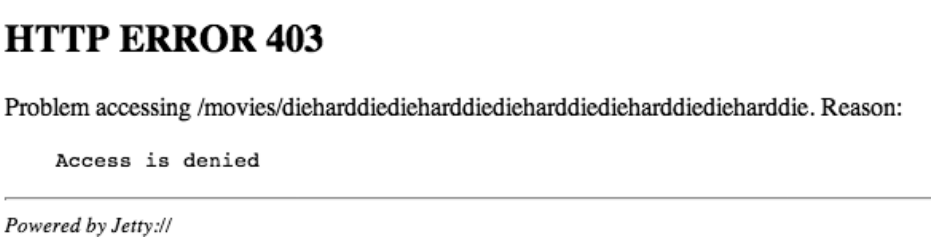


Figure 5-6. *Access denied page you get because the SpEL expression in a @PreAuthorize failed an evaluation*

As I said before, when you activate the use of pre and post annotations (with the `pre-post-annotations="enabled"` attribute in the `<security:global-method-security>` element), a new `AccessDecisionVoter` is added to the list of voters that the access decision manager will consult to decide whether or not to grant access to the secured method.

The voter of the concrete class `org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter` calls `org.springframework.security.access.expression.method.ExpressionBasedPreInvocationAdvice`, which creates a new `org.springframework.security.access.expression.method.MethodSecurityEvaluationContext`, which is an indirect implementation of Spring core's `org.springframework.expression.EvaluationContext` interface.

The `EvaluationContext` is the context in which the SpEL expressions will be evaluated. In Spring Security's method invocation support, an instance of `MethodSecurityEvaluationContext` is created, which extends the Spring Core's standard `org.springframework.expression.spel.support.StandardEvaluationContext` and adds support for accessing method parameters as variables in the SpEL expressions, as you did with the `#name` variable in the current example. The "`hasRole`" expression you are using is defined by the `org.springframework.security.access.expression.SecurityExpressionOperations` interface and its implementing class `org.springframework.security.access.expression.method.MethodSecurityExpressionRoot`, which is set as the root object of the `EvaluationContext` and against which all method calls in the expressions get evaluated.

In the previous chapter, you looked at the methods that the interface `SecurityExpressionOperations` expose to be used from the SpEL expressions for web security. For `@PreAuthorize` method security, these methods are the same. This means you can use the same expressions I talked about before, like `getAuthentication`, `isAnonymous`, `hasRole`, and so on.

Securing the Data Returned from a Method

Next let's study the `@PostAuthorize` annotation, which works mostly in the same way as `@PreAuthorize` but is used after the target method is invoked.

As you already should know, the `MethodSecurityInterceptor` wraps the call to the target method call between the `beforeInvocation` processing and an `afterInvocation` processing. This behavior is inherited from the `AbstractSecurityInterceptor` from which the `FilterSecurityInterceptor` also inherits. Figure 5-7 graphically refreshes this information.

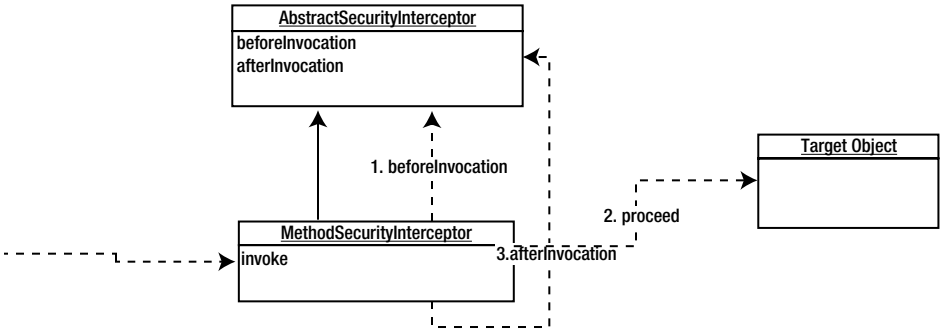


Figure 5-7. *MethodSecurityInterceptor wrapping the method call*

In the `afterInvocation` step (which is, in fact, a method in the `AbstractSecurityInterceptor`), an `AfterInvocationManager` implementation instance is consulted to make a final call in deciding if everything is OK to return the response to the requesting user. The `AfterInvocationManager` delegates to a list of `AfterInvocationProvider`(s) implementation instances to make the final decision.

When you configured the use of expressions for method security in the configuration XML, an instance of `PostInvocationAdviceProvider` was configured, at startup, in the list of `AfterInvocationProvider`(s) inside the `AfterInvocationManager`. This instance's `decide` method will be invoked. The first thing this method does is retrieve any post-invocation security attribute (`@PostFilter` or `@PostAuthorize`) from the general list of security attributes that exists in the target method. If one attribute is found, an instance of `ExpressionBasedPostInvocationAdvice` is called. Here, in this class, is where the expression context will be set up in order to evaluate the SpEL expression. The expression context is set up the same way it was for the `@PreAuthorize` use case. Figure 5-8 shows the main participants in this process.

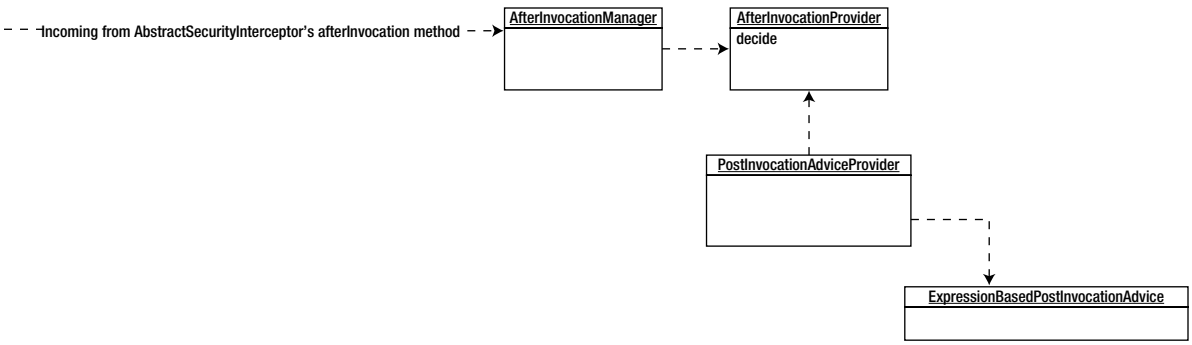


Figure 5-8. *@PostAuthorize processing behavior in afterInvocation*

Let's create another movie in the map you have in the MoviesServiceImpl class. In the static block, add the following: `MOVIES.put("two days in paris", new Movie("two days in paris ", "1000000"));`. Next let's remove the current `@PreAuthorize("#name.length() < 50 and hasRole('ROLE_USER')")` annotation from the `getMovieByName` method and instead add the following one: `@PostAuthorize("T(java.lang.Integer).parseInt(getReturnObject().budget) < 5000000")`. Then add the following import to the class: `import org.springframework.security.access.prepost.PostAuthorize;`. The expression is not difficult to understand. The first part, `T(java.lang.Integer.parseInt)`, is simply getting hold of the static method `parseInt` from the class `java.lang.Integer`. The parameter to `parseInt` is the property `"budget"` of the object returned from the model (the `Movie` object). Basically, the meaning of the whole annotation is "Post authorize if the returned movie's budget is less than 5000000."

After this is done, if you restart the application and visit the URL `http://localhost:8080/movies/two days in paris`, you should be able to access it without a problem. In this case, you don't even need to log in, as you are not making constraints on the authenticated user. Figure 5-9 shows the screen you see if you are following along. However, if you visit the URL `http://localhost:8080/movies/die hard`, you will get an "Access denied" message.

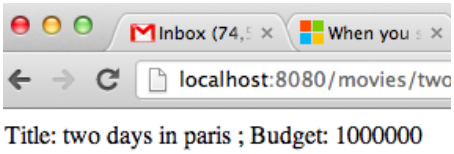


Figure 5-9. *Successfully accessing the movie*

■ **Note** Actually, you will get first the familiar login form. You get the login form here and not in the previous case because that is the standard way Spring Security handles Access Denied errors in the framework when there is not a fully authenticated user accessing the system. This means that before you log in, when an `AccessDeniedException` is thrown, Spring Security will note that there is no authenticated Authentication instance for the user trying to access the site. (There is probably only an `AnonymousAuthenticationToken` instance.) Spring Security will show the form, offering the user the opportunity to log in. After the user logs in, if he gets the `AccessDeniedException` again, this time an error message is shown informing him that he is not allowed to access the requested resource.

You can see from this example that you have access to the returned object from the method in the SpEL expression thanks to the `getReturnObject()` expression. This allows you to do any kind of validation you need on the returned object.

As I commented before, the expression context in which the SpEL expressions are executed uses an instance of `org.springframework.security.access.expression.method.MethodSecurityExpressionRoot` as the root object of the context against which the methods resolve. The method `getReturnObject()` is then defined in this class along with a few others, including `getThis()`, `getFilter()`, and all the ones inherited from `SecurityExpressionRoot`, as I said before.

Filtering Collections Sent and Returned from Methods

`@PreFilter` and `@PostFilter` annotations are most commonly used with access control list (ACL) security, but they can be used without them. Here, I will explain the general usage of these annotations and leave the ACL coverage for the next chapter.

The `@PreFilter` annotation allows you to filter collection parameters based on arbitrary expressions, removing certain elements so that they don't even arrive at the target secured method. Let's see how it works.

First, let's create the newMovies.jsp from Listing 5-7 in the WEB-INF/views folder, and then update the `MovieController` to look like Listing 5-8 and the `MoviesServiceImpl` to look like Listing 5-9. (Remember also to add the new method to the `MoviesService` interface like this: `"void addNewMovies(List<String> movies);"`). You use these new methods you are creating to create movies in the server by sending them in a CSV string.

Next, add the dependency to the pom.xml file:

```
<dependency>
          <groupId>org.springframework.security</groupId>
          <artifactId>spring-security-taglibs</artifactId>
          <version>3.1.2.RELEASE</version>
</dependency>
```

Next, restart the application, access the URL `http://localhost:8080/admin/movies`, and log in with `"admin"` `"admin"`. Then visit the URL `http://localhost:8080/movies/new` and you will see the form shown in Figure 5-10. Fill the text box with **aa,ff,badword**. Then click the submit button.

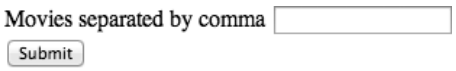


Figure 5-10. *Form with an input box to create movies*

Next, visit the URL `http://localhost:8080/movies/aa`. You get to the new movie's page, which shows you the title and budget. Figure 5-11 shows this page.

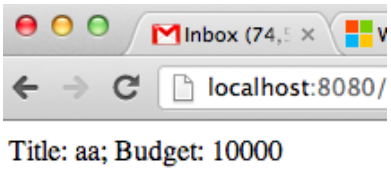


Figure 5-11. *A successfully stored movie*

Now visit the URL `http://localhost:8080/movies/badword`. An error page is returned because an `exception` was thrown. The error will make a reference to a null value in the expression—that is because the new movie with the title "badword" was never created, as it was filtered from the collection before getting to the method in `MoviesControllerImpl`. Figure 5-12 shows the important part of the error page, which says that it is trying to get the `budget` property from a null object. The movie doesn't exist. It is null.

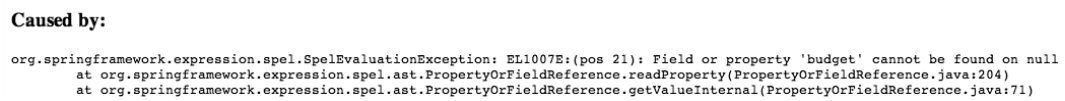


Figure 5-12. *Error page trying to access a nonexistent movie*

Figure 5-12 shows a stacktrace error page. Of course, you wouldn't want to show that in real-life, production-ready applications; instead, you would show a properly formatted, friendly error page and message. I haven't done anything in that regard because it isn't relevant to the concepts I want to show you.

You can see how it works. Inside the value of the annotation, you can use the special variable `filterObject` (which is really a getter method inside `MethodSecurityExpressionRoot`). This variable references each of the elements of the collection that is being passed to the method. Basically, when you are using this annotation, the expression handler iterates through the elements in the collection, sets the `filterObject` value in the expression root

as the current object in the iteration, and then evaluates the expression in the corresponding context. Next we see the listings for the jsp, controller and service updates referenced in the previous paragraphs. This is the code needed for showing a form to the user and allow him to create movies.

Listing 5-7. newMovies.jsp will show the the form for creating movies

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Terror movies</title>
</head>
<body>
    <form method="post">
        Movies separated by comma <input type="text" name="csvMovies"/><br/>
        <input type="submit"/>
    </form>
</body>
</html>
```

Listing 5-8. MovieController with New Methods for Adding New Movies

```
package com.apress.pss.terrormovies.controller;

import java.util.ArrayList;
import java.util.Arrays;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;

import com.apress.pss.terrormovies.model.Movie;
import com.apress.pss.terrormovies.service.MoviesService;

@Controller
@RequestMapping("/movies")
public class MovieController {

    @Autowired
    private MoviesService moviesService;

    @RequestMapping(method = RequestMethod.GET, value = "/{name}")
    @ResponseBody
```

Download from Wow! eBook <www.wowebook.com>

```
    public String getMovieByName(@PathVariable String name){
        Movie movie = moviesService.getMovieByName(name);
        return movie.toString();
    }

    @RequestMapping(method = RequestMethod.GET, value = "/new")
    public String showForm(){
        return "newMovies";
    }
    @RequestMapping(method = RequestMethod.POST, value = "/new")
    public String newMovies(@RequestParam String csvMovies){
        String[] movies = csvMovies.split(",");
        moviesService.addNewMovies(new ArrayList<String>(Arrays.asList(movies)));
        return "newMovies";
    }
}
```

Listing 5-9. MoviesServiceImpl with a New Method for Adding New Movies—Allowed Only for Admin Users and for MoviesThat Don't Contain Bad Words

```
package com.apress.pss.terrormovies.service;

import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.security.access.prepost.PostAuthorize;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.security.access.prepost.PreFilter;

import com.apress.pss.terrormovies.model.Movie;

public class MoviesServiceImpl implements MoviesService{

    private static final Map<String, Movie> MOVIES = new HashMap<String, Movie>();

    static{
        MOVIES.put("die hard", new Movie("Die Hard", "20000000"));
        MOVIES.put("two days in paris", new Movie("two days in paris ", "1000000"));
    }

    //@PreAuthorize("#name.length() < 50 and hasRole('ROLE_USER')")
    @PostAuthorize("T(java.lang.Integer).parseInt(getReturnObject().budget) < 5000000")
    public Movie getMovieByName(String name) {
        return MOVIES.get(name.toLowerCase());
    }

    @PreAuthorize("hasRole('ROLE_ADMIN')")
    @PreFilter("not filterObject.contains('badword')")
```

```
    public void addNewMovies(List<String> movies) {
        for(String movie: movies){
            MOVIES.put(movie, new Movie(movie, "10000"));
        }
    }

}
```

The `@PostFilter` annotation behaves mostly in the same way as the `@PreFilter` annotation, by filtering elements from a collection. The difference is that `@PostFilter` annotations are evaluated after the target method returns, and the SpEL expression in the annotation is evaluated against the returned collection from the target method. Note that I'm saying the "returned collection." This is a condition required for the `@PostFilter` annotation to work. As with the `@PreFilter` annotation, it applies only to collections and arrays. If the return type is neither of these two, the `DefaultMethodSecurityExpressionHandler` (which is the one evaluating the expressions) will throw an `IllegalArgumentException`.

Let's see it in action. Let's create a couple new methods and a new simple jsp file. The new functionality you'll add will allow you to show movies with a budget of over 5000000 only to ROLE_ADMIN users. For the rest of the users, these movies will be filtered out from the returned collection so that the jsp won't have access to them.

First, you add the method from Listing 5-10 to the `MoviesServiceImpl` class. (Remember to add the method signature to the corresponding `MovieService` interface as well—something like `Collection<Movie> getAllMovies();`.) This is the method where the `@PostFilter` annotation is added. You can see from Listing 5-10 that the SpEL expression is straightforward.

Listing 5-10. A Method That Returns Movies but Filters by Budget and Role

```
    @PreAuthorize("isFullyAuthenticated()")
    @PostFilter("hasRole('ROLE_ADMIN') or (hasRole('ROLE_USER') and  " +
"T(java.lang.Integer).parseInt(filterObject.budget) < 5000000)")
    public Collection<Movie> getAllMovies(){
        return new ArrayList<Movie>(MOVIES.values());
    }
```

Next, you add the method from Listing 5-11 to the `MovieController`. This method calls the service method and wraps the result in a model to be rendered in the view.

Listing 5-11. A MovieController New Method for Retrieving All the Movies

```
@RequestMapping(method = RequestMethod.GET, value = "/")
public ModelAndView getAllMovies(){
    ModelAndView mv = new ModelAndView("allMovies");
    mv.addObject("movies", moviesService.getAllMovies());
    return mv;
}
```

Finally, you create the jsp file that will support showing all the movies. You create the file allMovies.jsp from Listing 5-12 in the folder WEB-INF/views.

Listing 5-12. allMovies.jsp, Which Iterates Through the Movies and Prints Their Names

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags"%>
```

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Terror movies</title>
</head>
<body>
    <c:forEach items="${movies}" var="movie">
        ${movie.name}<br />
    </c:forEach>
</body>
</html>
```

Let's test this new configuration now.

Restart the application, and visit the URL `http://localhost:8080/movies/`. Log in with `"admin", "admin"`. You should see both movies' names shown on the page, as in Figure 5-13.

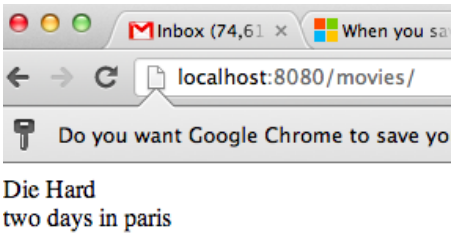


Figure 5-13. *An admin user can see all the movies*

Next, log out by visiting the URL `http://localhost:8080/j_spring_security_logout`.

■ **Note** In a real application, of course, you wouldn't ask your users to manually visit the URL `http://localhost:8080/j_spring_security_logout`. You would most likely have a Log Out link the user could click to log out. That link would point to the mentioned URL.

Visit the URL `http://localhost:8080/movies/`, log in with `"lucas", "fernandez"`. This time, you should only see the movie "two days in paris" because "die hard" was filtered out. Figure 5-14 shows this screen.

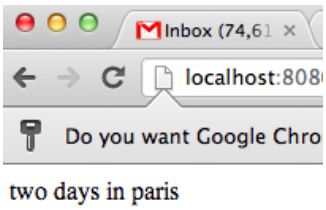


Figure 5-14. *A standard user cannot see movies with high budgets*

You can see how powerful security is when it's based on the SpEL expression. You can use this approach to define and combine different expressions to achieve security at different levels of granularity.

As I said before, the `@PreFilter` and `@PostFilter` annotations are normally used with ACLs and the `hasPermission` method. In the current example, I was explaining only the way the annotation works and how to use useful expressions to filter out elements from a collection. But the truth is that there is nothing particularly unsecure in returning the movies with budgets greater than $5 million to standard users in our current application.

Security Defined in XML

Annotation-based security is the most common solution that developers, using Spring Security, utilize when creating their business-layer security constraints. However, it is not the only option. As with most of the Spring suite of products, Spring Security security can be configured using XML configuration files.

There are supporters and detractors of both types of configurations (annotations and XML), and undoubtedly there are good and bad parts to either. One very good thing about the XML configuration for securing methods is that it can be applied to more than one method of more than one class at the same time using AspectJ pointcut expressions—the same way you would in the rest of Spring Security's suite of products.

The coverage of the syntax and options of AspectJ pointcut expressions is outside the scope of this book (You can learn more about them in *Pro Spring 3* by Clarence Ho and Rob Harrop (Apress, 2012) and *Spring Recipes, Second Edition* by Gary Mak, Daniel Rubio, and Josh Long (Apress, 2010)). Here, I will just give you a quick overview to help you use it in our example.

The most common pattern used in pointcut expressions is to match an undefined list of methods based on these methods' signatures. For example, the pointcut expression `execution(* com.apress.pss.MoviesService.*(..))` will match any method in the `MoviesService` interface. Let's use this expression in our example.

Let's say you want to make all your methods in `MoviesService` available only to users with `ROLE_ADMIN`. For that, let's remove all annotations from the `MoviesServiceImpl` class except for the `@PostAuthorize` one in `getMovieByName`, as you want to see how you can combine annotation security with XML configuration. This highlights another good way of establishing security. You can combine both methods (annotations and XML) to meet your particular set of requirements. So your `MoviesServiceImpl` class should now look like Listing 5-13.

Listing 5-13. MoviesServiceImpl Without Security Annotations (Except for One @PostAuthorize)

```
package com.apress.pss.terrormovies.service;

import java.util.ArrayList;
import java.util.Collection;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.springframework.security.access.prepost.PostAuthorize;
import com.apress.pss.terrormovies.model.Movie;

public class MoviesServiceImpl implements MoviesService{

    private static final Map<String, Movie> MOVIES = new HashMap<String, Movie>();

    static{
        MOVIES.put("die hard", new Movie("Die Hard", "20000000"));
        MOVIES.put("two days in paris", new Movie("two days in paris ", "1000000"));
    }
```

```
@PostAuthorize("T(java.lang.Integer).parseInt(getReturnObject().budget) < 5000000")
    public Movie getMovieByName(String name) {
        return MOVIES.get(name.toLowerCase());
    }

public void addNewMovies(List<String> movies) {
        for(String movie: movies){
            MOVIES.put(movie, new Movie(movie, "10000"));
        }
    }

public Collection<Movie> getAllMovies(){
        return new ArrayList<Movie>(MOVIES.values());
    }

}.
```

Next, you add the required XML configuration to guarantee that these methods are called only by a `ROLE_ADMIN` user. For that, you put the element `<security:protect-pointcut access="ROLE_ADMIN" expression="execution(* com.apress.pss.terrormovies.service.MoviesService.*(..))"/>` as a child of `<security:global-method-security>` in the applicationContext-security.xml file.

You need to add AspectJ dependency to the project. For that, you simply add the dependency from Listing 5-14 to your pom.xml.

Your application is configured now. If you restart the application and visit the URL `http://localhost:8080/movies/`, you will be redirected to the login screen. If you login with `"lucas", "fernandez"`, you will receive an "Access Denied" screen (the one you have seen many times before), as this user doesn't have `ROLE_ADMIN`. On the other hand, if you log in with `"admin", "admin"`, you are granted access to the page and will be able to see the movies on the page, which looks like Figure 5-13 again.

You can see how powerful the XML configuration can be in grouping many methods into a single expression to secure them all at the same time. The AspectJ expression language for pointcuts offers a lot of flexibility to determine which methods to match against. For example, you could specify all methods named `"doSecured"` (or any other name) in your application to be secured, or methods that receive a particular parameter type, or that return a particular type.

Listing 5-14. An AspectJ Maven Dependency

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.7.0</version>
</dependency>
```

Another way to handle security concerns in the XML configuration file is with the use of the element `<intercept-methods>` inside any bean in the application context to determine which methods from that bean must be secured and how they should be secured. This is, to me, the less convenient way of securing your methods, as it removes the clarity of the annotations, and at the same time, removes the flexibility of the pointcut XML expressions. Basically, it just adds XML verbosity in a place where some elegant annotations will do the same job.

Let's see this in action. Let's modify our `moviesService` bean in the applicationContext-security.xml file to look like Listing 5-15, and let's remove the `<protect-pointcut>` element you added in the previous example.

Listing 5-15. moviesService Bean with a Security Interceptor Embedded in the Bean Definition Itself

```
<bean id="moviesService" class="com.apress.pss.terrormovies.service.MoviesServiceImpl">
    <security:intercept-methods>
        <security:protect
            method="com.apress.pss.terrormovies.service.MoviesService.getAllMovies"
            access="ROLE_ADMIN" />
    </security:intercept-methods>
</bean>
```

This is all the configuration you need to do for securing that one method. If you restart the application now and visit the URL `http://localhost:8080/movies/`, the login form will appear on the page. If you log in with `"admin"`, `"admin"`, you be granted access to the page. You have effectively secured the method for only `ROLE_ADMIN` users. You can use wildcards in the method definitions to match more than one method at the same time. In the example, you could replace the `method` attribute of the `<security:protect>` element with the following to match all methods in the class: `method="com.apress.pss.terrormovies.service.MoviesService.*"`. With this new expression, all methods in the class require users to have the role `ROLE_ADMIN` in order to access them.

Security Without a Web Layer

Even if this chapter has focused on securing the business layer of applications through the use of method-level security, you are still working in the context of a web application. This is the standard model in which Spring Security is used. A web front end takes care of the authentication process, and a combination of URL and method security rules and constraints takes care of the authorization process.

The web layer, through the use of all the filters you reviewed in the previous chapter, give you a lot of out-of-the-box functionality for authenticating users into the application. Your only job is to configure certain beans in the application context so that you can leverage all this functionality.

However, there should be no reason why you can't leverage the rest of the Spring Security infrastructure in a non-web environment. Let's take a look at how to do this in a simple command-line-based application. You will reuse most of your current code in the service layer, but you'll create a thin command-line layer you can use to interact with the system. First, let's see what you need to do to make this work.

Spring Security's web support is focused mainly in the authentication part of security—the part where the user presents his credentials and is correctly identified by the system. Most of the authorization part is handled by the `AbstractSecurityInterceptor` and all of its helping collaborators. As I have explained in this chapter, `MethodSecurityInterceptor` is one of the `AbstractSecurityInterceptor`'s concrete implementations and does not, at all, depend on the application being web based.

The authentication process' only concern, as I just said, is to take the username and credentials of a user (normally from a login form, but it can be from many different places), match these credentials to the saved users, and then create a new `Authentication object (or, more accurately, one of its many implementations)` that will carry all the necessary information of the just logged-in user, including her roles in the application.

So this part is the one you need to change for your command-line interface to work. You need a way of providing the username and credentials, and then create an `Authentication`'s implementation object instance when the authentication infrastructure matches this user.

Thanks to Spring Security's highly modular system, there are many places where you can plug in your new command-line "login" infrastructure, and then leverage most of the logic already provided by the framework.

In the following examples, I will cover the basic functionality of logging in to the application with a fully authenticated user, accessing a secured method, and then logging out. Everything will be simple and concise and will serve to illustrate the idea. First, you create the simple interface from Listing 5-16 in the package `com.apress.pss.terrormovies.access` of your application.

Listing 5-16. Interface to be Used from the Command-Line Client

```
package com.apress.pss.terrormovies.access;

public interface AccessOperations {
    void login(String username, String password);
    void logout();
    String executeOperation(String beanName, String method);
}
```

You can see three simple methods: one to log in, one to log out, and one to execute a random method (parameterless) in a Spring bean. The idea is that this method will be secured with Spring Security, so you should get familiar behavior when trying to access it with different levels of an authenticated user.

Next, let's define the class that will actually receive the command-line requests and translate them to invocations on this interface. This will be your main class, and it will take care of starting the Spring application context. Remember that previously the application context was loaded and started by a Servlet Listener configured in the web.xml file. Listing 5-17 shows this `Main` class, which I created in the package `com.apress.pss.terrormovies.standalone`.

Listing 5-17. Main Class for the Standalone Command-Line-Based Application

```
package com.apress.pss.terrormovies.standalone;

import java.io.Console;

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

import com.apress.pss.terrormovies.access.AccessOperations;

public class Main {

    private AccessOperations operations;

    public static void main(String[] args) {
        Main main = new Main();
        main.loadContext();
        main.initCommandInput();
    }

    private void initCommandInput() {
        Console c = System.console();
        if (c == null) {
            System.err.println("No console.");
            System.exit(1);
        }
        String command = "";
        do {
            try {
                command = c.readLine("Insert command: ");
                interpretAndExecute(command);
            } catch (Exception e) {
                e.printStackTrace();
            }
```

```
        } while (!command.equals("exit"));
    }

    private void interpretAndExecute(String command) {
        if (command.equals("logout")) {
            operations.logout();
        } else if (command.startsWith("login")) {
            String userAndPassword = command.split(" ")[1];
            String user = userAndPassword.split(":")[0];
            String password = userAndPassword.split(":")[1];
            operations.login(user, password);
        } else {
            String[] beanAndMethod = command.split(" ");
            String returned = operations.executeOperation(beanAndMethod[0], beanAndMethod[1]);
            System.out.println("Returned: "+returned);
        }
    }

    private void loadContext() {
        ApplicationContext context = new ClassPathXmlApplicationContext(
                "applicationContext-security.xml");
        operations = context
                .getBean("accessOperations", AccessOperations.class);
    }

}
```

That is a very simple class, which simply loops through accepting command-line commands. It supports only three types of commands, and you will see them here. But first, let's define the class that will take care of the actual authentication process and that will serve as the entry for invoking Spring bean methods. Listing 5-18 shows this class, which is an implementation of the `AccessOperations` interface presented in Listing 5-16.

Listing 5-18. AccessOperationsImpl That Allows Login and Logout and by Default Uses an Anonymously Logged-on User

```
package com.apress.pss.terrormovies.access;

import org.springframework.beans.BeansException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.security.authentication.AnonymousAuthenticationToken;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.authority.AuthorityUtils;
import org.springframework.security.core.context.SecurityContextHolder;

public class AccessOperationsImpl implements AccessOperations, ApplicationContextAware {

    private AuthenticationManager authenticationManager;
    private ApplicationContext context;
```

```
    public AccessOperationsImpl(){
        SecurityContextHolder.setStrategyName("MODE_GLOBAL");
        initAnonymous();

    }

    public void login(String username, String password) {
        UsernamePasswordAuthenticationToken authRequest =
            new UsernamePasswordAuthenticationToken(username, password);
           Authentication authentication = authenticationManager.authenticate(authRequest);
        SecurityContextHolder.getContext().setAuthentication(authentication);
    }

    public void logout() {
        SecurityContextHolder.clearContext();
        initAnonymous();
    }

    public String executeOperation(String beanName, String method) {
        Object bean =  context.getBean(beanName);
        Object result = null;
        try {
            result = bean.getClass().getMethod(method).invoke(bean);
        } catch (Exception e) {
            e.printStackTrace();
                      return null;
        }
        return result.toString();
    }

    public void setApplicationContext(ApplicationContext context)
            throws BeansException {
        this.context = context;
    }

    public void setAuthenticationManager(AuthenticationManager authenticationManager) {
        this.authenticationManager = authenticationManager;
    }

    private void initAnonymous() {
        AnonymousAuthenticationToken auth =
             new AnonymousAuthenticationToken("anonymous",
                 "anonymousUser",AuthorityUtils.createAuthorityList("ROLE_ANONYMOUS"));
        SecurityContextHolder.getContext().setAuthentication(auth);
    }
}
```

This class should also be straightforward if you have followed through the book. It handles all the authentication processing you require for your application. When it is instantiated by the Spring loading process, it begins by setting a Security Context Strategy to `MODE_GLOBAL`. This means that for the whole application there will be only the one and the same `SecurityContext object instance (one SecurityContextImpl, to be more exact, with the implementation)`. This strategy is only good in this kind of standalone (without a server part) application, where only one user will be using the application at any given time.

The next thing the code does is initialize the security context with an anonymous `Authentication` object (an instance of `org.springframework.security.authentication.AnonymousAuthenticationToken`), the same way `AnonymousAuthenticationFilter` does in web-based applications.

Next, you need to configure this object as a bean in the application context. I copied the applicationContext-security.xml from the WEB-INF directory into the src/main/resources folder so that it can be found in the classpath when you run the application as a standalone application. Then, in this file, I created the required bean. Listing 5-19 shows the whole applicationContext-security.xml file

Listing 5-19. applicationContext-security.xml for the Standalone Command-Line-Based Application

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns=http://www.springframework.org/schema/beans
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">

    <security:global-method-security
        secured-annotations="enabled" jsr250-annotations="enabled"
        pre-post-annotations="enabled">
        <!-- <security:protect-pointcut access="ROLE_ADMIN"
            expression="execution(* com.apress.pss.terrormovies.service.MoviesService.*(..))" /> -->
    </security:global-method-security>

    <security:http auto-config="true" />

    <security:authentication-manager alias="authenticationManager">
        <security:authentication-provider
            user-service-ref="inMemoryUserServiceWithCustomUser" />
    </security:authentication-manager>

    <bean id="expressionHandler"
        class="com.apress.pss.terrormovies.security.CustomWebSecurityExpressionHandler" />

    <bean id="inMemoryUserServiceWithCustomUser"
        class="com.apress.pss.terrormovies.spring.CustomInMemoryUserDetailsManager">
        <constructor-arg>
            <list>
                <bean class="com.apress.pss.terrormovies.model.User">
                    <constructor-arg value="admin" />
                    <constructor-arg value="admin" />
                    <constructor-arg>
                        <list>
                            <bean

class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                                <constructor-arg value="ROLE_ADMIN" />
                            </bean>
                        </list>
                    </constructor-arg>
                    <constructor-arg value="Scarioni" />
                    <constructor-arg value="19" />
```

Download from Wow! eBook <www.wowebook.com>

```
                </bean>
                <bean class="com.apress.pss.terrormovies.model.User">
                    <constructor-arg value="paco" />
                    <constructor-arg value="tous" />
                    <constructor-arg>
                        <list>
                            <bean

class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                                <constructor-arg value="ROLE_USER" />
                            </bean>
                        </list>
                    </constructor-arg>
                    <constructor-arg value="Miranda" />
                    <constructor-arg value="20" />
                </bean>
                <bean class="com.apress.pss.terrormovies.model.User">
                    <constructor-arg value="lucas" />
                    <constructor-arg value="fernandez" />
                    <constructor-arg>
                        <list>
                            <bean

class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                                    <constructor-arg value="ROLE_VIP" />
                            </bean>
                            <bean

class="org.springframework.security.core.authority.SimpleGrantedAuthority">
                                <constructor-arg value="ROLE_USER" />
                            </bean>
                        </list>
                    </constructor-arg>
                    <constructor-arg value="Silva" />
                    <constructor-arg value="20" />
                </bean>
            </list>
        </constructor-arg>
    </bean>

    <bean id="moviesService" class="com.apress.pss.terrormovies.service.MoviesServiceImpl">
        <security:intercept-methods>
            <security:protect
                method="com.apress.pss.terrormovies.service.MoviesService.*"
                access="ROLE_ADMIN" />
        </security:intercept-methods>
    </bean>

    <bean id = "accessOperations" class="com.apress.pss.terrormovies.access.AccessOperationsImpl">
        <property name="authenticationManager" ref="authenticationManager"/>
    </bean>
</beans>
```

All is ready to go. You should know how to run a simple java application with a main class, but just to make it easier, you can use this simple command to run the application in the Maven environment and set up its dependencies: `mvn exec:java -Dexec.mainClass="com.apress.pss.terrormovies.standalone.Main"`. Let's exercise the program and see how it works step by step. Command outputs and exceptions will be truncated if necessary to show only the relevant values.

- 1. Try to call a bean's method that is secured. You get an `AccessDeniedException`.

```
Insert command: moviesService getAllMovies
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at com.apress.pss.terrormovies.access.AccessOperationsImpl.executeOperation
(AccessOperationsImpl.java:40)
    at com.apress.pss.terrormovies.standalone.Main.interpretAndExecute(Main.java:47)
    at com.apress.pss.terrormovies.standalone.Main.initCommandInput(Main.java:30)
    at com.apress.pss.terrormovies.standalone.Main.main(Main.java:17)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.
java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run(ExecJavaMojo.java:297)
    at java.lang.Thread.run(Thread.java:722)
Caused by: org.springframework.security.access.AccessDeniedException: Access is denied
```

- 2. Send a login with incorrect credentials. You get a `BadCredentialsException`.

```
Insert command: login admin:afmin
org.springframework.security.authentication.BadCredentialsException: Bad credentials
    at org.springframework.security.authentication.dao.DaoAuthenticationProvider.
            additionalAuthenticationChecks(DaoAuthenticationProvider.java:67)
    at org.springframework.security.authentication.dao.
AbstractUserDetailsAuthenticationProvider.
            authenticate(AbstractUserDetailsAuthenticationProvider.java:149)
    at org.springframework.security.authentication.ProviderManager.
            authenticate(ProviderManager.java:156)
    at com.apress.pss.terrormovies.access.AccessOperationsImpl.
           login(AccessOperationsImpl.java:27)
    at com.apress.pss.terrormovies.standalone.Main.interpretAndExecute(Main.java:44)
    at com.apress.pss.terrormovies.standalone.Main.initCommandInput(Main.java:30)
    at com.apress.pss.terrormovies.standalone.Main.main(Main.java:17)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.
         invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run(ExecJavaMojo.java:297)
    at java.lang.Thread.run(Thread.java:722)
```

- 3. Log in with a correct user, with role ROLE_ADMIN.

```
Insert command: login admin:admin
Insert command:
```

- 4. Try to call the bean's method again. This time it works, as you are logged in correctly.

```
Insert command: moviesService getAllMovies
Returned: [Title: Die Hard; Budget: 20000000, Title: two days in paris ; Budget: 1000000]
```

- 5. Log out.

```
Insert command: logout
Insert command:
```

- 6. Try to execute the bean method, and again you get `AccessDeniedException`, as you have already logged out.

```
Insert command: moviesService getAllMovies
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.
        invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at com.apress.pss.terrormovies.access.AccessOperationsImpl.
        executeOperation(AccessOperationsImpl.java:40)
    at com.apress.pss.terrormovies.standalone.Main.interpretAndExecute(Main.java:47)
    at com.apress.pss.terrormovies.standalone.Main.initCommandInput(Main.java:30)
    at com.apress.pss.terrormovies.standalone.Main.main(Main.java:17)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.
        invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.
       invoke(DelegatingMethodAccessorImpl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:601)
    at org.codehaus.mojo.exec.ExecJavaMojo$1.run(ExecJavaMojo.java:297)
    at java.lang.Thread.run(Thread.java:722)
Caused by: org.springframework.security.access.AccessDeniedException: Access is denied
```

This flow should be familiar, as it is a simplified version of the flow that happens during Spring Security web-based authentication.

In step 1, you are trying to access a secured resource (a method, in this case) without being fully authenticated in the system. The application uses, by default, an Anonymous `Authentication`, which is not enough for accessing this particular resource.

In step 2, you are trying to log in to the system with the wrong credentials. Appropriately, Spring Security is throwing a `BadCredentialsException` indicating that this is, in fact, the case. You can see in the stacktrace the process the authentication request follows until the exception is thrown by the `DaoAuthenticationProvider`.

In step 3, you are logging in with a correct username and password set. This time, you get no output in the command line. What is happening is simply that the `DaoAuthenticationProvider` is verifying that the password matches the saved credentials this time, so it is not throwing any exceptions. The `AbstractUserDetailsAuthenticationProvider`

then creates a successful `Authentication` instance (an instance of `UsernamePasswordAuthenticationToken`) under the hood, and that is the one you are storing in the `SecurityContext` in your login method.

In step 4, when you try to access the secured resource again, the framework under the hood will notice that there is an `Authentication` instance stored in the `SecurityContext` that has the required authority needed to access the resource. This is enforced in the standard way you studied before. The `MethodSecurityInterceptor` works together with the `AccessDecisionManager` and the `AccessDecisionVoter` to decide that the logged-in user has permission to access the requested method.

In step 5, you simply log out of the application. The logout process here is simple, you clear the `SecurityContext`, and then initialize it again with an `Anonymous Authentication`.

Step 6 is the same as step 1. I am simply demonstrating in this step that the logout process had the required effect in the application.

You can see how this simple application is leveraging most of the functionality from Spring Security. You needed only a thin layer of authentication, working as an entry point, to get access to the full power of the security features that Spring Security offers. This means you can integrate Spring Security into your Swing, AWT, command-line applications, among other types with relative ease. It is always good to know that there is a tried and tested framework that can help you address the different concerns in your application. In the case of security, you should consider Spring Security independently of the application stack you have, as you can probably leverage a lot of functionality from it.

Using AspectJ AOP instead of Spring AOP

Spring Security offers the possibility to integrate with AspectJ aspects instead of standard Spring AOP. The advantages of AspectJ over Spring are many and I will start to explain them by first explaining briefly what AspectJ is itself.

As we have explained to a certain degree, Aspect Oriented Programming (AOP) is a programming paradigm that allows to separate the crosscutting concerns of an application into their own dedicated modules.

AspectJ is a dedicated Java based implementation of AOP managed by eclipse.org. AspectJ extends Java with an aspect-dedicated syntax and a new compiler that allows for the weaving of aspects to your applications.

Talking about weaving, what does it mean?. Weaving in AOP parlance, is simply the process in which the aspect functionality is combined with the core business functionality to produce the final working entity that deals with both concerns (crosscutting and core business). Weaving can happen at a few points in an application.

- It can happen at compile time where a special compiler is able to combine the source code of the business class with the code of the aspect to generate a new class containing the required functionality.
- It can happen at compile-link time. In this case the business class and the aspect are compiled to bytecode and then a special 'linker' is able to combine the resulting bytecodes into the aspect enhanced class.
- It can happen at load time. In this case the bytecode is combined as classes are being loaded into the Java Virtual Machine (JVM). In AspectJ, an application will normally have a configuration file specifying the Aspects and will need to be started with a special Java agent (–javaagent http://docs.oracle.com/javase/6/docs/api/java/lang/instrument/package-summary.html).
- It can happen at runtime. In this case, which is the case we have been looking at so far with Spring AOP, aspect functionality is combined with the business functionality by composing objects together at runtime. This is commonly achieved with the use of proxies as we have seen before.

While Spring AOP focus exclusively in runtime weaving, AspectJ supports the other kinds of weaving. This makes AspectJ a more powerful solution (as you are not just limited to intercepting public non static method calls) at the cost of more complexity. In the majority of cases when using Spring, Spring AOP will be everything you need to use.

Spring Security brings with it a module named spring-security-aspects. This is a very small module which contains only one Aspect and one AspectJ configuration file for load time weaving of the said Aspect. Listing 5-20 shows the Aspect source code.

Listing 5-20. AspectJ Security Aspect

```
package org.springframework.security.access.intercept.aspectj.aspect;

import org.springframework.beans.factory.InitializingBean;
import org.springframework.security.access.annotation.Secured;
import org.springframework.security.access.prepost.*;
import org.springframework.security.access.intercept.aspectj.AspectJCallback;
import org.springframework.security.access.intercept.aspectj.AspectJMethodSecurityInterceptor;

/**
 * Concrete AspectJ aspect using Spring Security @Secured annotation
 * for JDK 1.5+.
 *
 * <p>
 * When using this aspect, you <i>must</i> annotate the implementation class
 * (and/or methods within that class), <i>not</i> the interface (if any) that
 * the class implements. AspectJ follows Java's rule that annotations on
 * interfaces are <i>not</i> inherited. This will vary from Spring AOP.
 *
 * @author Mike Wiesner
 * @author Luke Taylor
 * @since 3.1
 */
public aspect AnnotationSecurityAspect implements InitializingBean {

    /**
     * Matches the execution of any public method in a type with the Secured
     * annotation, or any subtype of a type with the Secured annotation.
     */
    private pointcut executionOfAnyPublicMethodInAtSecuredType() :
        execution(public * ((@Secured *)+).*(..)) && @this(Secured);

    /**
     * Matches the execution of any method with the Secured annotation.
     */
    private pointcut executionOfSecuredMethod() :
        execution(* *(..)) && @annotation(Secured);

    /**
     * Matches the execution of any method with Pre/Post annotations.
     */
    private pointcut executionOfPrePostAnnotatedMethod() :
        execution(* *(..)) && (@annotation(PreAuthorize) || @annotation(PreFilter)
                || @annotation(PostAuthorize) || @annotation(PostFilter));

    private pointcut securedMethodExecution() :
        executionOfAnyPublicMethodInAtSecuredType() ||
        executionOfSecuredMethod() ||
        executionOfPrePostAnnotatedMethod();
```

```
    private AspectJMethodSecurityInterceptor securityInterceptor;

    Object around(): securedMethodExecution() {
        if (this.securityInterceptor == null) {
            return proceed();
        }

        AspectJCallback callback = new AspectJCallback() {
            public Object proceedWithObject() {
                return proceed();
            }
        };

        return this.securityInterceptor.invoke(thisJoinPoint, callback);
    }

    public void setSecurityInterceptor(AspectJMethodSecurityInterceptor securityInterceptor) {
        this.securityInterceptor = securityInterceptor;
    }

    public void afterPropertiesSet() throws Exception {
        if (this.securityInterceptor == null) {
            throw new IllegalArgumentException("securityInterceptor required");
        }
    }

}
```

In the previous Listing, you can see the use of AspectJ syntax (which I won't explain here as it is outside the scope of the book). The source code is well commented so you can get a sense of the way it works. Remember that pointcuts define places that will match a particular piece of code where we want the aspect to execute, and in the code snippet from Listing 5-20, in every pointcut, it is explained where will it match. Worth noting is the use of the AspectJMethodSecurityInterceptor which is yet another concrete implementation of the AbstractSecurityInterceptor that we have seen before.

We will be using compile time weaving in our example, and we will prove the use of AspectJ by securing a private method in our service layer. Something we know is not possible with standard Spring AOP.

We will be starting from scratch for this example so let's do some of the usual work. First we execute, from the command line, the following command to create our project: `mvn archetype:generate -DgroupId=com.apress.pss -DartifactId=aspectj-project -DarchetypeArtifactId=maven-archetype-webapp`.

Then we replace the pom.xml file with the one from Listing 5-21.

Listing 5-21. pom.xml file for AspectJ example

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
       <modelVersion>4.0.0</modelVersion>
       <groupId>com.apress.pss</groupId>
       <artifactId>aspectj-project</artifactId>
       <packaging>war</packaging>
       <version>1.0-SNAPSHOT</version>
       <name>aspectj-project Maven Webapp</name>
       <url>http://maven.apache.org</url>
```

```
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
        </dependency>

        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.5</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-core</artifactId>
            <version>3.1.2.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>3.1.2.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>3.1.2.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>3.0.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjweaver</artifactId>
            <version>1.7.0</version>
        </dependency>

        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>1.7.1</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
```

```
                <artifactId>spring-security-aspects</artifactId>
                <version>3.1.3.RELEASE</version>
            </dependency>
        </dependencies>
        <build>
            <finalName>aspectj-project</finalName>
            <plugins>
                <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-compiler-plugin</artifactId>
                    <configuration>
                        <compilerVersion>1.6</compilerVersion>
                        <fork>true</fork>
                        <source>1.6</source>
                        <target>1.6</target>
                    </configuration>
                </plugin>

                <plugin>
                    <groupId>org.codehaus.mojo</groupId>
                    <artifactId>aspectj-maven-plugin</artifactId>
                    <version>1.4</version>
                    <executions>
                        <execution>
                            <id>compile</id>
                            <configuration>
                                <source>1.6</source>
                                <target>1.6</target>
                                <verbose>false</verbose>
                                <outxml>true</outxml>
                                <aspectLibraries>
                                    <aspectLibrary>
        <groupId>org.springframework.security</groupId>
                                                    <artifactId>spring-
security-aspects</artifactId>
                                    </aspectLibrary>
                                </aspectLibraries>
                            </configuration>
                            <goals>
                                <goal>compile</goal>
                            </goals>
                        </execution>
                        <execution>
                            <id>test-compile</id>
                            <configuration>
                                <source>1.6</source>
                                <target>1.6</target>
                                <verbose>false</verbose>
                                <aspectLibraries>
                                    <aspectLibrary>
```

```
<groupId>org.springframework.security</groupId>
                                                        <artifactId>spring-security-
aspects</artifactId>
                                                    </aspectLibrary>
                                                </aspectLibraries>
                                            </configuration>
                                            <goals>
                                                <goal>test-compile</goal>
                                            </goals>
                                        </execution>
                                    </executions>
                                    <dependencies>
                                        <dependency>
                                            <groupId>org.aspectj</groupId>
                                            <artifactId>aspectjrt</artifactId>
                                            <version>1.7.1</version>
                                        </dependency>
                                    </dependencies>
                                </plugin>
                                <plugin>
                                    <groupId>org.mortbay.jetty</groupId>
                                    <artifactId>jetty-maven-plugin</artifactId>
                                    <version>8.1.1.v20120215</version>
                                    <configuration>
                                        <connectors>
                                            <connector implementation="org.eclipse.jetty.server.nio.
SelectChannelConnector">
                                                <port>8080</port>
                                                <maxIdleTime>60000</maxIdleTime>
                                            </connector>
                                        </connectors>
                                    </configuration>
                                </plugin>
                            </plugins>
                        </build>
</project>
```

In the previous Listing is important to note the introduction of the AspectJ dependencies and more importantly, the use of the aspectj-maven-plugin. This plugin is the one that will take care of the process of weaving the Aspects into the application. Note the element <aspectLibraries> in the plugin definition. This element will find the already compiled aspect library that we want to weave into our application in the compilation process. Note how I am using the spring-security-aspects module that I have just mentioned before.

The next thing we will do is to write our main configuration files: web.xml, aspect-example-servlet.xml and applicationContext-security.xml which you can see in Listings 5-22, 5-23 and 5-24 respectively. These files will all live in the WEB-INF directory.

Listing 5-22. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
       <listener>
         <listener-class>
            org.springframework.web.context.ContextLoaderListener
         </listener-class>
       </listener>
       <context-param>
              <param-name>contextConfigLocation</param-name>
              <param-value>
                 /WEB-INF/applicationContext-security.xml
             </param-value>
       </context-param>

       <servlet>
              <servlet-name>aspect-example</servlet-name>
              <servlet-class> org.springframework.web.servlet.DispatcherServlet </servlet-class>
              <load-on-startup>1</load-on-startup>
       </servlet>
       <servlet-mapping>
              <servlet-name>aspect-example</servlet-name>
              <url-pattern>/</url-pattern>
       </servlet-mapping>
       <filter>
              <filter-name>springSecurityFilterChain</filter-name>
              <filter-class>
                     org.springframework.web.filter.DelegatingFilterProxy
              </filter-class>
       </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Listing 5-23. aspect-example-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:security="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
http://www.springframework.org/schema/beans
```

Download from Wow! eBook <www.wowebook.com>

```
http://www.springframework.org/schema/beans/spring-beans-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-3.1.xsd">

	<context:component-scan base-package="com.apress.pss.aspects.controllers" />
	<mvc:annotation-driven />

	<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
		<property name = "prefix" value="/WEB-INF/views/" />
		<property name = "suffix" value=".jsp" />
	</bean>

</beans>
```

Listing 5-24. applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
	xmlns:security="http://www.springframework.org/schema/security"
	xmlns:context="http://www.springframework.org/schema/context"
	xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">
	<security:http auto-config="true" />
	<security:authentication-manager>
		<security:authentication-provider>
			<security:user-service>
				<security:user authorities="ROLE_USER" name="car"
					password="scarvarez" />
			</security:user-service>
		</security:authentication-provider>
	</security:authentication-manager>
	<context:component-scan base-package="com.apress.pss.aspects" />
	<security:global-method-security
		secured-annotations="enabled" mode="aspectj">
	</security:global-method-security>

</beans>
```

In this last file (applicationContext-security.xml) note how we have added the 'mode="aspectj"' to the the <global-method-security> element. This element will be picked up by the class `GlobalMethodSecurityBeanDefinitionParser` when the application is starting up, and will define the Aspect `AnnotationSecurityAspect` and inject the proper interceptor to it.

Next let's define a couple of classes. A controller which you can see in Listing 5-25 and a service which you can see in Listing 5-26.

Listing 5-25. Controller TheController in package com.apress.pss.aspects.controllers

```
package com.apress.pss.aspects.controllers;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

import com.apress.pss.aspects.Service;

@Controller
@RequestMapping("/hello")
public class TheController {

	@Autowired
	private Service service;

	@RequestMapping("/")
	public String doSomething(){
		 return service.methodA();
	}
}
```

Listing 5-26. Service Service.java in package com.apress.pss.aspects

```
package com.apress.pss.aspects;

import org.springframework.security.access.annotation.Secured;

@org.springframework.stereotype.Service
public class Service {
	public String  methodA() {
		return methodB();
	}

	@Secured("ROLE_USER")
	private String methodB() {
		return "AspectJ Secured String";
	}
}
```

Make sure to save both previous classes in their corresponding package. In the `Service` listing you can see that I have secured the private method `methodB` which is called by the public method in the same class `method.` This, as you know, wouldn't have any effect if using Spring AOP.

That's it. Let's run the example. From the root of the project, and as you have done so many times now, execute mvn clean install jetty:run.

After the application starts, if you visit the URL `http://localhost:8080/hello/` you will be shown the login form that we have seen so many times now. If you login with username 'car' password 'scarvarez', you will be able to access the page shown in Figure 5-15 which is the expected result (As Spring MVC is trying to translate the returned String "AspectJ Secured String" into a view name jsp which it can't find).

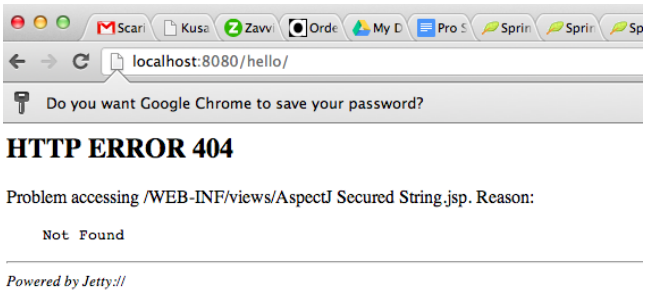


Figure 5-15. *page accessed with the right ROLE_USER credentials*

This is very cool. We have successfully secured a private Method. If you were to debug the `Service` class in both methods (method and methodB) you would note that there is not a typical Spring AOP Proxy around any of them (as we are not using Spring AOP but AspectJ). Figures 5-16 and 5-17 show a snapshot of both methods being debugged.

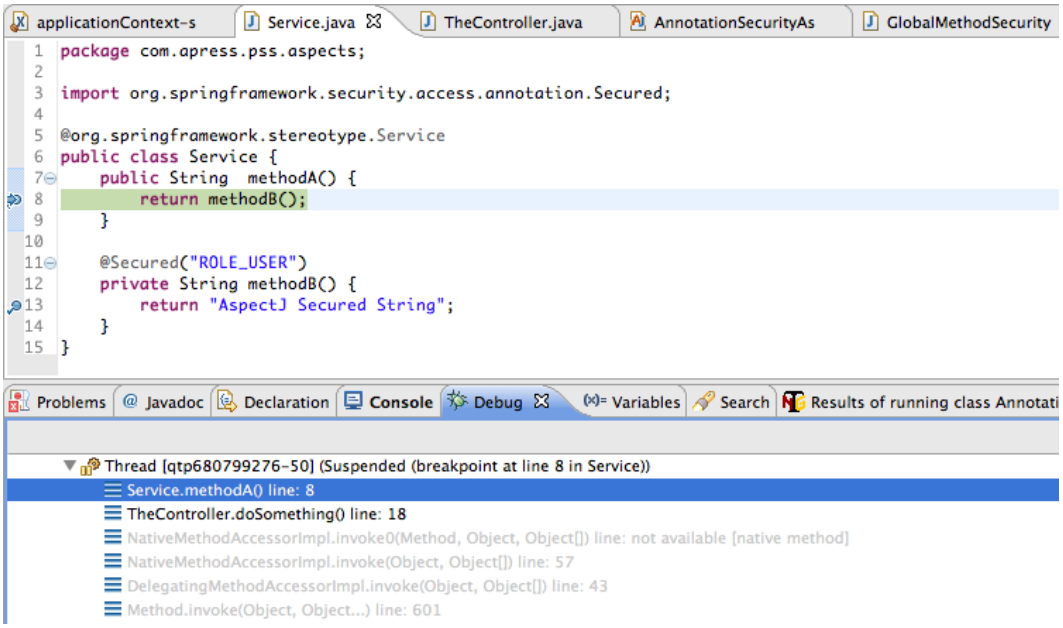
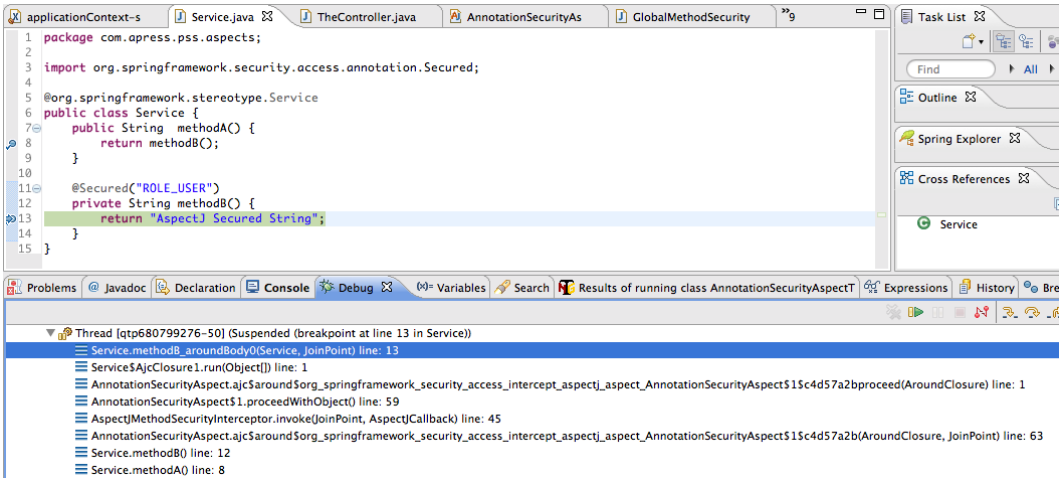


Figure 5-16. *methodA being debugged. No presence of Proxy. Controller calls Service directly*




```
1  package com.apress.pss.aspects;
2
3  import org.springframework.security.access.annotation.Secured;
4
5  @org.springframework.stereotype.Service
6  public class Service {
7      public String  methodA() {
8          return methodB();
9      }
10
11     @Secured("ROLE_USER")
12     private String methodB() {
13         return "AspectJ Secured String";
14     }
15 }
```

Figure 5-17. *methodB being debugged. No presence of Proxy but AspectJ trickery*

That's it on the cover of AspectJ with Spring Security. Another great tool to have at your disposal.

Summary

In this chapter, I covered Spring Security's support for method-level security. You should have a good idea of when to use this type of security and how to apply it to different layers in your applications. You should also know the different ways in which you can achieve method-level security, including using annotations, SpEL, and XML. You saw that Spring Security can be used without a web layer, even when the standard use case is for web applications. I also covered the use of Spring Security with the powerful AspectJ library to secure private methods. Also, you should have a clear understanding of the difference between the authentication and the authorization process.

CHAPTER 6

Configuring Alternative Authentication Providers

One of Spring Security's strongest points is that you can plug different authentication mechanisms into the framework. Spring Security was built to create, as much as possible, a pluggable architecture model, where different things can be plugged into the framework in an easy and unobtrusive way. In the authentication layer, this means that an abstraction exists that takes care of this part of the security process. This abstraction comes in the form, mainly, of the `AuthenticationProvider` interface, but it also is supported by specific security Servlet filters and user details services.

Among the different authentication mechanisms I will cover are the following:

- LDAP
- OpenID
- X.509
- JAAS
- Database

Most of this chapter deals with explaining how each of these authentication systems work, independently of Spring Security. Although this gives you certain key details, it won't be an in-depth explanation. Of course, you will see how Spring Security implements each of these authentication mechanisms, and you will see that they have many things in common when it comes to the parts of Spring Security they use.

Database-Provided Authentication

Database-provided authentication works almost exactly the same way as with the memory-provided authentication users. The only difference is that the users are stored in the database and not in memory. This happens at runtime when you define them in the application context configuration file.

When you are using database-authentication mechanisms, the `AuthenticationProvider` implementation doesn't need to change at all—it's still the `DaoAuthenticationProvider` you used for the in-memory user authentication. As you may remember, this `AuthenticationProvider` implementation is based on using a `UserDetailsService` abstraction to retrieve the users, so the difference here is the `UserDetailsService` implementation you will use.

To configure the provisioning of users from the database is pretty simple, as you will see. We will clean up and build on the application from Chapter 2. Let's start.

The first thing you do is define the database schema you need in order to make the authentication mechanism work. The tables you need to make the authentication work are shown in Figure 6-1.

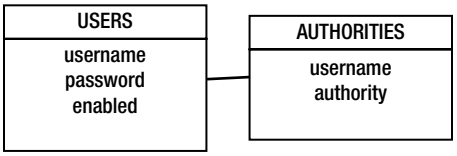


Figure 6-1. *Simple DB schema needed to support database authentication*

Figure 6-1 shows a simple schema model for supporting authentication backed by a database. Just by looking at the tables, you should be able to see how they work. It is basically a one-to-one mapping from the in-memory implementation you have been using so far. In the `USER` table, you store the user details—mainly, the username and the password. In the `AUTHORITIES` table, you store the relationship between the usernames and the granted authorities for that member—for example, `ROLE_MEMBER`.

Figure 6-2 shows an extended default option you can use to define groups and establish authorities related to those groups instead of to individual users.

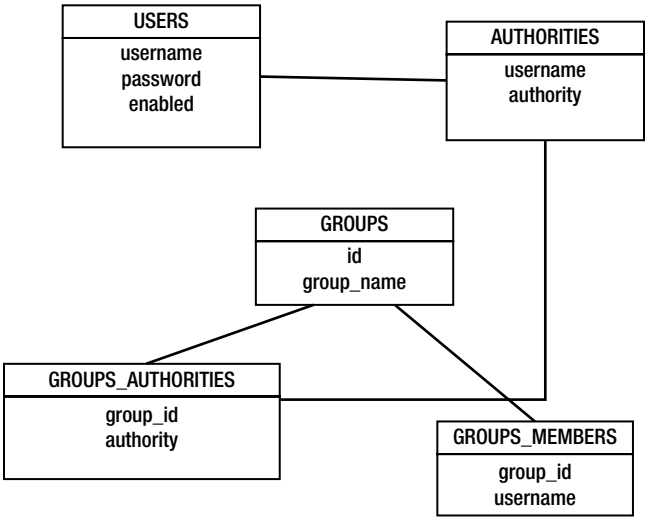


Figure 6-2. *Database-backed authentication scheme with groups*

Figure 6-2 is a bit more complex than Figure 6-1; however, it is still very straightforward. This time, the schema allows you to create named groups and establish authorities belonging to these groups. At the same time, users now can belong to groups as well, meaning that users can inherit the authorities defined for the groups to which they belong. For example, if there is a group named "Administrators" in the `GROUPS` table and there are two authorities (`ROLE_ADMIN` and `ROLE_USER`) defined in the `GROUP_AUTHORITIES` table that map to the group "Administrators," then a member belonging to group "Administrators" (which exists in the table `GROUP_MEMBERS`) will have the authorities `ROLE_ADMIN` and `ROLE_USER` inherited from the group.

By default, the group mechanism is not activated in `UserDetailsService`. To activate it, you need to configure a `groupAuthoritiesByUsernameQuery` attribute in the corresponding `<jdbc-user-service>` in the configuration file, as you will see later in the example.

■ **Note** For this example project and the rest of the examples in this chapter, unless otherwise noted, you will start with an application configured as follows.

1. Create an application with the following command:

```
mvn archetype:generate -DgroupId=com.apress.pss
     -DartifactId=name-of-the-app -DarchetypeArtifactId=maven-archetype-webapp.
```

Remember to replace the `artifactId` property with a unique name for your application.

Listing 6-1. web.xml to be used in all applications in this chapter

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
    <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
           /WEB-INF/applicationContext-security.xml
        </param-value>
    </context-param>

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

2. Replace the generated web.xml file with the one in Listing 6-2.

Listing 6-2. Starting applicationContext-security.xml to be used in all examples in this chapter

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
```

```
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">
    <security:http auto-config="true">
        <security:intercept-url pattern="/*" access="ROLE_USER" />
    </security:http>
</beans>
```

3. Create a file named `applicationContext-security.xml` with the contents of Listing 6-3 in the generated WEB-INF folder.

Listing 6-3. Simple Servlet definition

```
package com.apress.pss.servlets;

import java.io.IOException;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns={"/hello"})
public class HelloWorldServlet extends HttpServlet {
    private static final long serialVersionUID = 2218168052197231866L;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response){
        try {
            response.getWriter().write("Hello World");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

4. Create a simple Servlet in the package `com.apress.pss.servlets` with the contents from Listing 6-4.

Listing 6-4. Bootstrap pom.xml with some Spring Security configuration and Jetty

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.pss</groupId>
    <artifactId>authentication-xxx</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>authentication-cas Maven Webapp</name>
    <url>http://maven.apache.org</url>
```

```
<dependencies>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>javax.servlet-api</artifactId>
        <version>3.0.1</version>
    </dependency>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.5</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-core</artifactId>
        <version>3.1.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-config</artifactId>
        <version>3.1.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-web</artifactId>
        <version>3.1.3.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.1.1</version>
    </dependency>
    <dependency>
        <groupId>commons-codec</groupId>
        <artifactId>commons-codec</artifactId>
        <version>1.3</version>
    </dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>keytool-maven-plugin</artifactId>
            <configuration>
                <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
                <dname>cn=localhost</dname>
                <keypass>jetty8</keypass>
                <storepass>jetty8</storepass>
                <alias>localhost</alias>
                <keyalg>RSA</keyalg>
            </configuration>
        </plugin>
```

```
                <plugin>
                    <groupId>org.mortbay.jetty</groupId>
                    <artifactId>jetty-maven-plugin</artifactId>
                    <version>8.1.1.v20120215</version>
                    <configuration>
                        <connectors>
                            <connector implementation="org.eclipse.jetty.server.nio.
                            SelectChannelConnector">
                                <port>8080</port>
                                <maxIdleTime>60000</maxIdleTime>
                            </connector>
                            <connector implementation="org.eclipse.jetty.server.ssl.
                            SslSocketConnector">
                                <port>8443</port>
                                <maxIdleTime>60000</maxIdleTime>
                                <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
                                <password>jetty8</password>
                                <keyPassword>jetty8</keyPassword>
                            </connector>
                        </connectors>
                        <scanIntervalSeconds>10</scanIntervalSeconds>
                        <stopKey>foo</stopKey>
                        <stopPort>8999</stopPort>
                        <contextPath>/</contextPath>
                    </configuration>
                </plugin>
            </plugins>
        </build>
    </project>
```

5. Replace the generated `pom.xml` with the one from Listing 6-5. Remember to change the `artifactId` name with the name of your application.

   Listing 6-5. HSQLDB Maven dependency

   ```
   <dependency>
       <groupId>org.hsqldb</groupId>
       <artifactId>hsqldb</artifactId>
       <version>2.2.8</version>
   </dependency>
   ```

Let's get the example going. We'll use an HSQL database, so you need to configure its JDBC dependencies in the `pom.xml` file as Listing 6-5 shows.

Next let's modify the configuration file `applicationContext-security.xml` to include the configuration changes needed to support database-driven user authentication. Listing 6-6 shows the `applicationContext-security.xml` file.

Download from Wow! eBook <www.wowebook.com>

Listing 6-6. applicationContext-security.xml that uses a database driven UserDetailsService

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

    <security:http auto-config="true">
        <security:intercept-url pattern="/*" access="ROLE_SCARVAREZ_MEMBER" />
        <security:form-login/>
    </security:http>

    <security:authentication-manager>
        <security:authentication-provider>
            <security:jdbc-user-service data-source-ref="dataSource" />
        </security:authentication-provider>
    </security:authentication-manager>

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

In the file from Listing 6-6, you can see that you are using the namespace `<jdbc:>`to define a new embedded datasource through the element `<embedded-database>`. Embedded datasources are a new feature introduced in Spring 3.0. You can use them to define different kinds of embedded datasources (basically, in-memory datasources that run within the Java process where they are used), such as HSQL and Derby. HSQL is the default. You can also see here that in the tag you are also allowed to specify SQL script locations you want to execute when the datasource is started up. The scripts execute in sequential order from the top down—in our example, `security-schema.sql` is executed first, and then `users.sql` is executed. The SQL files you are specifying here allow you to create a simple database schema to support the configuration of users and authorities the way you defined them previously in the section.

Using this embedded datasource is very convenient for examples of this type and unit testing your application, but most likely you won't use them in production environments. For production environments, you will use full database solutions, such as MySQL, PostgreSQL, Oracle, and others.

Creating the Basic Tables

Next you create the two SQL files you are referencing in the configuration file and put them in the root of the classpath. Listing 6-7 shows the file `security-schema.sql`, and Listing 6-8 shows the file `users.sql`. Later, you will see groups coming into the picture as well.

Listing 6-7. security-schema.sql defines the needed security database schema for storing users and authorities

```
CREATE TABLE USERS(USERNAME VARCHAR_IGNORECASE(50) NOT NULL PRIMARY KEY,
                   PASSWORD VARCHAR_IGNORECASE(50) NOT NULL,
                   ENABLED BOOLEAN NOT NULL);

CREATE TABLE AUTHORITIES(
                         USERNAME VARCHAR_IGNORECASE(50) NOT NULL PRIMARY KEY,
                         AUTHORITY VARCHAR_IGNORECASE(50) NOT NULL,
                         CONSTRAINT FK_AUTHORITIES_USERS
                              FOREIGN KEY(USERNAME) REFERENCES USERS(USERNAME));

CREATE TABLE GROUPS(
                    id BIGINT NOT NULL PRIMARY KEY,
                    GROUP_NAME VARCHAR_IGNORECASE(50) NOT NULL);

CREATE TABLE GROUP_MEMBERS(
                           group_id BIGINT NOT NULL,
                           username VARCHAR_IGNORECASE(50) NOT NULL);

CREATE TABLE GROUP_AUTHORITIES(
                              group_id BIGINT NOT NULL,
                              authority VARCHAR_IGNORECASE(50) NOT NULL);
```

Listing 6-8. users.sql users that be used in the application with their authorities

```
INSERT INTO USERS VALUES('car','scarvarez',true);
INSERT INTO AUTHORITIES VALUES('car','ROLE_SCARVAREZ_MEMBER');
COMMIT;
```

Everything should be set up now, so start the application:

```
mvn clean install jetty:run
```

You should be able to log in and access the URL `http://localhost:8080/hello` only if you log in with the username **car** and the password **scarvarez**. Figures 6-3 and 6-4 show the login page and the "Hello World" page that I get in my execution of the example.



Figure 6-3. *Login screen*

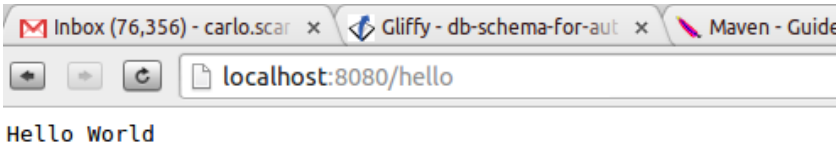


Figure 6-4. *The "Hello World" page shown after a successful authentication*

What is happening here is pretty simple and similar to what you have been working with up until now. When you define the element `<security:jdbc-user-service>` in the configuration file `applicationContext-security.xml`, Spring Security instantiates a different kind of `UserDetailsService` at startup than the one used for in-memory user details.

The instantiated `UserDetailsService` will be an `org.springframework.security.provisioning.JdbcUserDetailsManager`. This instance will be injected into the `DaoAuthenticationProvider` for retrieving users from a JDBC datastore.

Using Groups

To use groups now, let's modify the `users.sql` file a little bit and add the lines from Listing 6-9 to it. Those lines effectively create a group and put the user "car" into that group. You should also remove the SQL where you insert into the `AUTHORITIES` table the role for user "car".

Listing 6-9. The users.sql lines used to create a group and put members into group

```
INSERT INTO GROUPS VALUES(1,'GROUP_MEMBERS_SCARV');
INSERT INTO GROUP_MEMBERS VALUES(1,'car');
INSERT INTO GROUP_AUTHORITIES VALUES(1,'ROLE_SCARVAREZ_MEMBER');
```

Now, to activate the use of groups in the `UserDetailsService` you need to set the property `enableGroups` to `true` in the `JdbcDaoImpl` implementation; however, for some reason, the `<jdbc-user-service>` namespace element currently doesn't support the setting of this simple property directly. What it does support is the setting of the attribute `group-authorities-by-username-query`, which allows you to specify the query to retrieve the groups from the database schema. Setting this attribute automatically sets the property `enableGroups` to `true`. The other option, of course, is not to use the XML namespace and instead use the standard bean definition and set the property. You are going to use the first option and make the `<jdbc-user-service>` in the `applicationContext-security.xml` look like Listing 6-10.

Listing 6-10. <jdbc-user-service> in the configuration file that specifies the group-authorities-by-username-query attribute

```
<security:jdbc-user-service data-source-ref="dataSource"
      group-authorities-by-username-query="select g.id, g.group_name, ga.authority
                  from groups g, group_members gm, group_authorities ga
                  where gm.username = ? and g.id = ga.group_id and g.id = gm.group_id"/>
```

Consider the query you specified in Listing 6-10; the `group-authorities-by-username-query` attribute is extracted directly from the class file `JdbcDaoImpl` (the query string, that is). It is configured in the `DEF_GROUP_AUTHORITIES_BY_USERNAME_QUERY` constant in that file (`JdbcDaoImpl`), and it is the default query used by the class to retrieve the groups and authorities.

If you restart the application now, you should see the same behavior as before. This time, however, the behavior internally is different because the group query is the one being executed to retrieve the authorities for the user. The code that makes this choice in the `JdbcDaoImpl` is very simple, and you can see it in Listing 6-11. The property that is used in the second `if` condition is set automatically by Spring Security when starting up, particularly in the class `JdbcUserServiceBeanDefinitionParser` when you set the `group-authorities-by-username-query` attribute as I explained before. The `enableAuthorities` property in the first `if` is automatically set to `true` in the `JdbcDaoImpl` class itself.

Listing 6-11. Code inside JdbcDaoImpl that queries by group or username

```
if (enableAuthorities) {
   dbAuthsSet.addAll(loadUserAuthorities(user.getUsername()));
}

if (enableGroups) {
   dbAuthsSet.addAll(loadGroupAuthorities(user.getUsername()));
}
```

The advantage of using groups is that it offers a new level of organizing users into the same category and does not relate them directly to particular authorities. Instead, it relates the whole group to the authorities. For example, you would normally say that all administrators belong to the Administrator Group, and this group has many authorities related to it, like `ROLE_ADMIN`, `ROLE_USER`, `ROLE_OPERATOR`, and others. All these authorities can be grouped into the same conceptual group, and at the same time, all users can be grouped into their respective groups. This means that not all users need to reference all these authorities. They simply need to be made part of the group. Of course, a user can belong to more than one group at the same time.

Using Existing Schemas

`JdbcDaoImpl`, by default, is configured to use the database schema and queries you have looked at already. However, the schema configuration is flexible, and you can adapt your existing database user schema (if any) to be used by Spring Security JDBC `UserDetails` support instead of writing a custom schema just for Spring Security. Of course, certain concepts need to exist in your current schema, like a "user" abstraction, an "authorities" or "role" definition, and a "group" abstraction if you will be also using groups. If you have those abstractions in place in your database, accessing the information from Spring Security's JDBC support is straightforward. The two attributes (`authorities-by-username-query` and `group-authorities-by-username-query`) in the element `<jdbc-user-service>` allow you to specify the exact SQL that will be used to retrieve the authorities for a particular username when not using groups or when indeed using groups.

You saw the example in the previous section of how to use the `group-authorities-by-username-query` attribute. The `authorities-by-username-query` is configured similarly, and it should be a query that returns a pair of columns in its resultset. The first one represents the username, and the second one represents a particular authority. For example, the default query looks like this:

```
select username,authority from authorities where username = ?
```

The other attribute you need to change in the `<jdbc-user-service>` element is `users-by-username-query`. Here you use the query needed to retrieve users using their username. The query needs to return three columns in a record: `username`, `password`, and `enabled`.

Let's try this out with a quick example. You will change just some files from the example we are currently working on. First, you will change your `security-schema.sql` and `users.sql` files to something that doesn't really match the default values Spring Security expects. So replace the contents of those files with the contents of Listing 6-12 and Listing 6-13, respectively.

Listing 6-12. A security-schema.sql that doesn't match Spring Security defaults

```
CREATE TABLE PEOPLE(
                    NAME VARCHAR_IGNORECASE(50) NOT NULL PRIMARY KEY,
                    KEY VARCHAR_IGNORECASE(50));

CREATE TABLE ROLES(
                   NAME VARCHAR_IGNORECASE(50) NOT NULL PRIMARY KEY,
                   ROLE VARCHAR_IGNORECASE(50) NOT NULL,
                   CONSTRAINT FK_AUTHORITIES_USERS FOREIGN KEY(NAME)
                   REFERENCES PEOPLE(NAME));
COMMIT;
```

Listing 6-13. A users.sql that doesn't match Spring Security defaults

```
INSERT INTO PEOPLE VALUES('car','scarvarez');
INSERT INTO ROLES VALUES('car','ROLE_SCARVAREZ_MEMBER');
COMMIT;
```

Next, replace the `<security:jdbc-user-service>` element from the file `applicationContext-security.xml` with the one from Listing 6-14.

Listing 6-14. JDBC User Service with custom queries

```
<security:jdbc-user-service
        data-source-ref="dataSource"
        authorities-by-username-query="select name, role from roles where name = ?"
        users-by-username-query="select name,key,1 from people where name = ?" />
```

Now restart the application, log in with the username **car** and the password **scarvarez**, and access the URL `http://localhost:8080/hello`. You should be able to reach the page in exactly the same way as before.

LDAP Authentication

The Lightweight Directory Access Protocol (LDAP) is an application-level, message-oriented protocol used for storing and accessing information in the way of an accessible tree-like directory. A directory, in general, is simply an organized datastore that allows for easy queries in its particular domain. For example, a TV Guide is a directory that allows you to find TV shows easily, and a phone book is a directory that provides easy access to phone numbers.

LDAP allows the storage of very different kinds of information in a directory. Probably the most widely known use of LDAP-like structures is the Microsoft Windows Active Directory system. Other LDAP systems are widely used to store the corporate user databases of many companies that serve as the centralized user store.

LDAP is not an easy system to understand, and I will try to explain it along with the example that I develop in this section. I will use the same code as in the previous section, but modify it as needed to work with LDAP instead of database authentication. Remember that in the previous section, I offered a bootstrap application to start working in all the examples in this chapter—including this one.

The first thing to do is configure your users in the LDAP directory. To do this, you need to understand the LDAP Information Model, which defines the type of data you can store in your directory.

The data in LDAP is defined by entries, attributes, and values. An *entry* is the basic unit of information in the directory and commonly represents an entity from the real world, like a user. Entries are normally defined by a particular object class. Each entry in the directory has an identification known as a Distinguished Name (or, more commonly, DN). Each entry in the directory also has a set of *attributes* that describe different things about the entry. Each attribute has a type and one or more *values*.

We have to define the data we will need. We will use users, groups, and credentials as we have been doing so far. Commonly in LDAP, the user entry definition is known as *People*, so we will use that name to define the user entries. Our user will also use the standard LDAP object class `person` to define its attributes.

Installing and Configuring LDAP

We will be using an ApacheDS LDAP implementation, which is a pure Java implementation. You need to download and install the ApacheDS server from its web site: `http://directory.apache.org/`. You should download and install both the Server (ApacheDS) and the Studio (Apache Directory Studio). The installation process of both tools is straightforward and shouldn't give you any issues. Consult the official web page for directions.

The Apache Directory Studio allows you to access and query the server. The Apache Directory Studio is an Eclipse application built of plugins adapted to access LDAP servers. Regarding the ApacheDS server, when it is installed (at least on a Mac), it will be started up automatically; however, you also have the option of running the ApacheDS server as an embedded server within the application process instead of having a standalone, independent LDAP server. We will be using the standalone server in the example.

After installing the Server and Studio, you need to connect to the Server from the Studio. To do that, go to the File menu in the Studio, select New, and then select LDAP Connection, as shown in Figure 6-5. In the LDAP connection form, enter the values to connect to the local ApacheDS server as shown in Figure 6-6. You can see the host is "localhost" and the port is 10389. Click Next to finish the connection configuration.

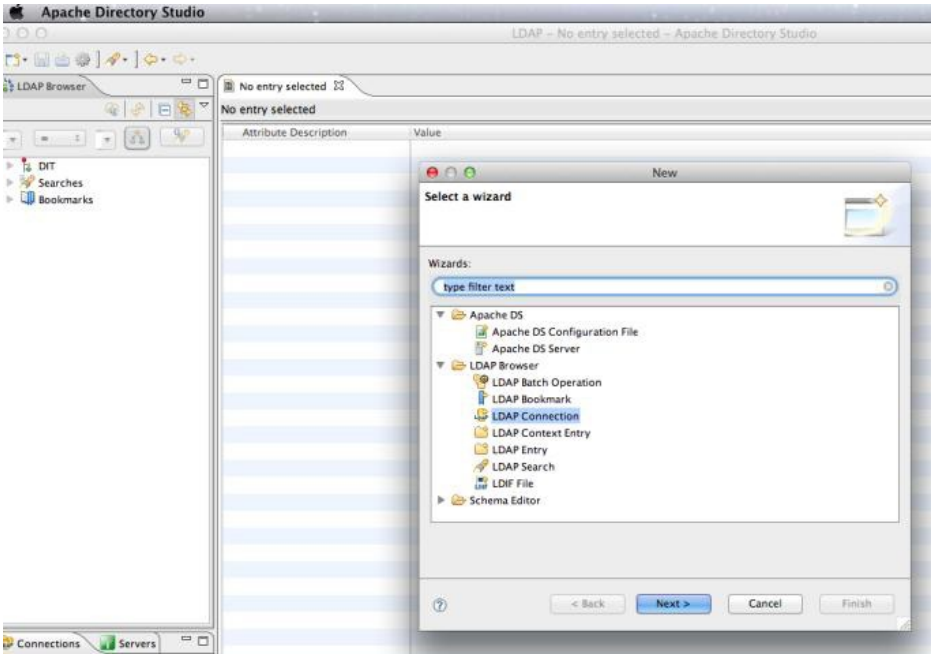


Figure 6-5. *ApacheDS Directory Studio connecting to the LDAP server*

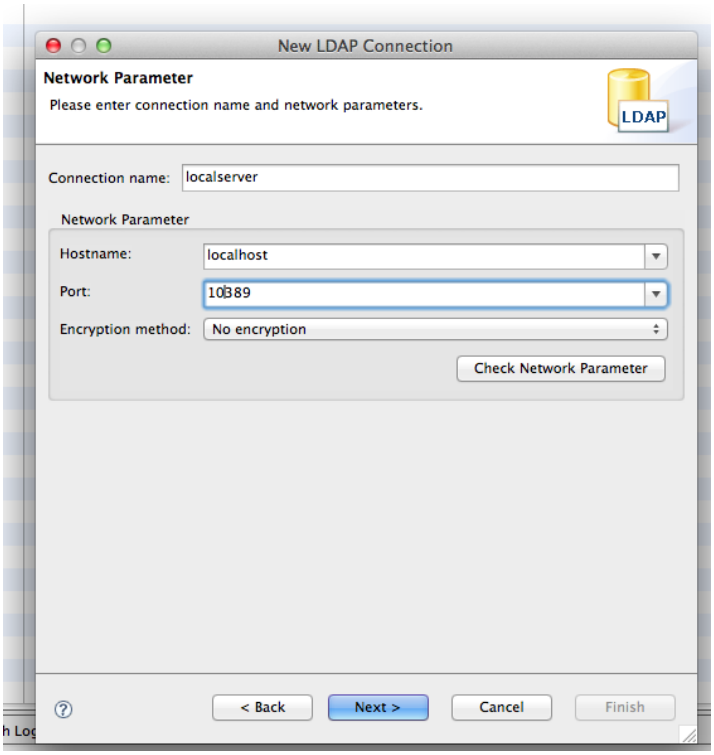


Figure 6-6. *Connecting to the local ApacheDS server*

Once the connection to the ApacheDS server is established, you need to create a context entry representing the top-level entry in the local directory. To do that, you right click on the left panel's `Root DSE` element and then select New ➤ New Context Entry in the contextual menu, as shown in Figure 6-7.

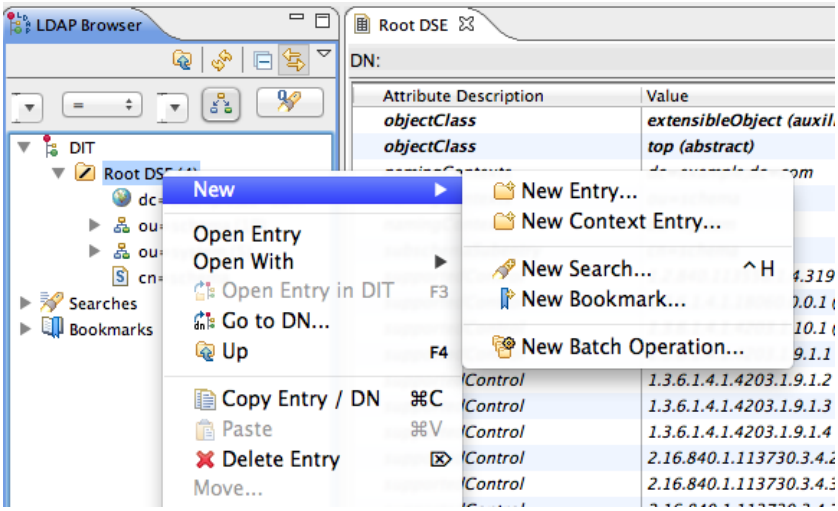


Figure 6-7. *Creating a new context entry top-level entry*

In the first form in New Context Entry, you choose to create an entry from scratch. Then, from the list of available object classes in the left panel, you select `dNSDomain` and click Add. The idea here is that you are creating the top-level entry from which all other entries derive. You end up with something like Figure 6-8. Then click Next. In the next input form, use `dc=example,dc=com` as the DN name of the context entry, and click Next. These values (`example` and `com`) are the default values for the partition and the top-level context entry included in the ApacheDS server. These values also partially identify every entry you create in the directory because, as I said before, LDAP follows a hierarchical structure that builds names for entries on top of its parent entries.

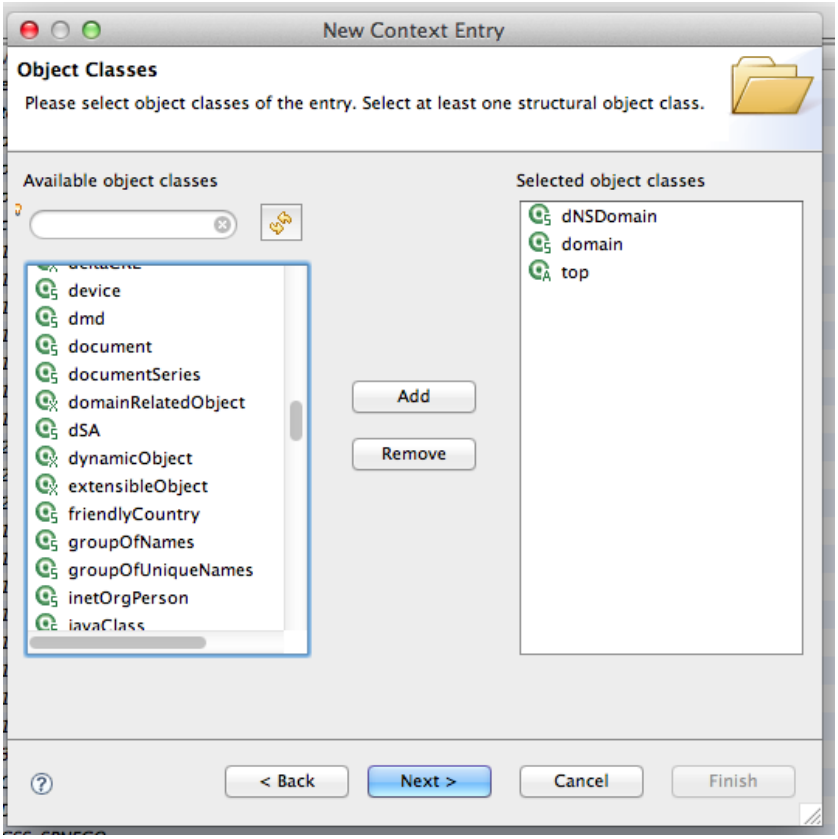


Figure 6-8. *Selecting the dNSDomain object class for the context entry*

The next step is to import your users into the LDAP server. I will be importing a couple of users using an LDIF file. An LDIF file is a text file that uses LDIF formatting, a standard format for describing directory entries in LDAP. It allows you to import and export your directory data into or from another LDAP directory in a standard way or just to create new data or modify existing data. You use it here to import the data with your users. Listing 6-5 shows the LDIF file you will import. You can name this file whatever you like. I will call it `users.ldif`.

Listing 6-5. LDIF file with the two users you want to import into the LDAP directory

```
dn: ou=groups,dc=example,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups
```

```
dn: ou=people,dc=example,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=car,ou=people,dc=example,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Carlo Scarioni
sn: Carlo
uid: car
userPassword: scarvarez

dn: uid=mon,ou=people,dc=example,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Monica Alvarez
sn: Monica
uid: mon
userPassword: scarvarez

dn: cn=administrators,ou=groups,dc=example,dc=com
objectclass: top
objectclass: groupOfNames
cn: administrators
ou: administrator
member: uid=mon,ou=people,dc=example,dc=com

dn: cn=users,ou=groups,dc=example,dc=com
objectclass: top
objectclass: groupOfNames
cn: users
ou: user
member: uid=car,ou=people,dc=example,dc=com
```

You can see in Listing 6-5 the hierarchical nature of the directory and how everything inherits the DN `dc=example,dc=com`. You can also see how the different entries use different standard object classes. You have created two groups: administrators and users. You also established that "*car*" is a member of the "users" group and "*mon*" is a member of the "administrators" group. The password for both users is "scarvarez." In this example, they are shown in plain text, although they could easily be stored in encrypted form. Graphically, the hierarchy is simple enough and looks like Figure 6-9.

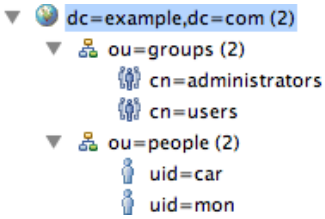


Figure 6-9. *LDAP hierarchy showing the structure you created with the LDIF file*

To import the file from Listing 6-5, from Apache Directory Studio, right-click on the left panel in the newly created context entry. Then select Import ➤ LDIF Import. In the input form, shown in Figure 6-10, select and find the file where you stored the content from Listing 6-5 (`users.ldif`, in my case), and click Finish.

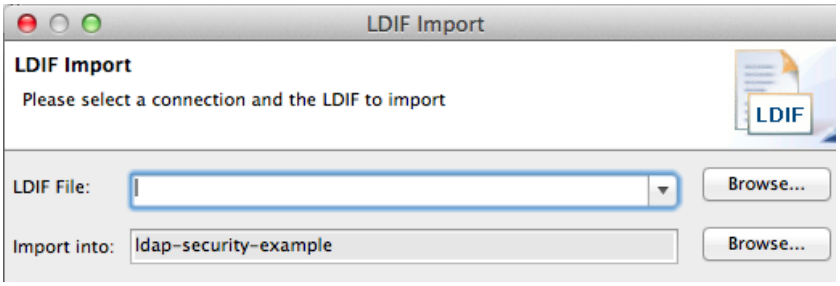


Figure 6-10. *Importing the LDIF file with Apache Directory Studio*

The next thing you need to do is configure the example application to be able to connect to the LDAP server and query the information stored on it. You have a clean application at the moment, with only the bootstrap components. You will start to add the required functionality now.

First, you add the Spring Security LDAP dependency to the `pom.xml` file. The dependency is shown in Listing 6-6.

Listing 6-6. Spring Security LDAP dependency

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-ldap</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>
```

Next you need to create the actual Spring configuration. In Listing 6-7, I show you the configuration file `applicationContext-security.xml` and then I shall explain how it works.

Listing 6-7. applicationContext-security.xml file, an example of Spring Security configuration for LDAP-based authentication and credential storage

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


Download from Wow! eBook <www.wowebook.com>

```
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

    <security:http auto-config="true">
        <security:intercept-url pattern="/*" access="ROLE_ADMINISTRATORS" />
        <security:form-login/>
    </security:http>

    <security:authentication-manager>
        <security:ldap-authentication-provider user-dn-pattern="uid={0},ou=people"
                  group-search-base="ou=groups" group-search-filter="(member={0})"/>
    </security:authentication-manager>

<security:ldap-server url="ldap://localhost:10389/dc=example,dc=com" />

</beans>
```

Listing 6-7 shows all the configuration needed to make your application work with LDAP-based authentication. If you restart the application now with that configuration, you should be able to access the URL `http://localhost:8080/hello` only if you log in with the username **mon** and the password **scarvarez**. If you log in with the username **car** and the password **scarvarez**, you get an "access denied" message.

The configuration is not that difficult to understand. The element `<ldap-server>` allows you to configure the LDAP server you will be connecting to from the application, and here, you are specifying the local ApacheDS server you configured in previous steps. You also specify the Distinguished Name you will be using as part of the connection URL in this same `<ldap-server>` element. This is basically the root path of your directory, as you specified before. Remember that all your entries will be relative to the domain name `dc=example,dc=com`.

The element `<ldap-authentication-provider>` is where the meat of the configuration is. You can see that instead of configuring a different `UserDetailsService` as in the case of database-based authentication, you are configuring a whole new `AuthenticationProvider`. Here you are specifying three attributes: `user-dn-pattern`, `group-search-base`, and `group-search-filter`.

The `user-dn-pattern` specifies how the user will be authenticated in the directory. Here it will replace the pattern `{0}` with the passed username for the user and then attempt a bind operation against the LDAP server. *Binding* is the standard way of authenticating against an LDAP server. In a bind operation, a user provides a DN and some credentials. Then the LDAP server checks this information and, if it is a valid authentication information, the LDAP server marks that the user is authenticated while the connection remains open or the user authenticates again.

The attribute `group-search-base` specifies the base for where it will start to search for group membership. It will look inside the hierarchy starting in this base for the groups defined.

The attribute `group-search-filter` specifies the attribute for a group in which the members of the group will be found. The pattern `{0}` will be replaced by the DN of the user.

This is how it works. Figure 6-11 shows a graphical description of the process, up to the point of creating a successful `Authentication` object.

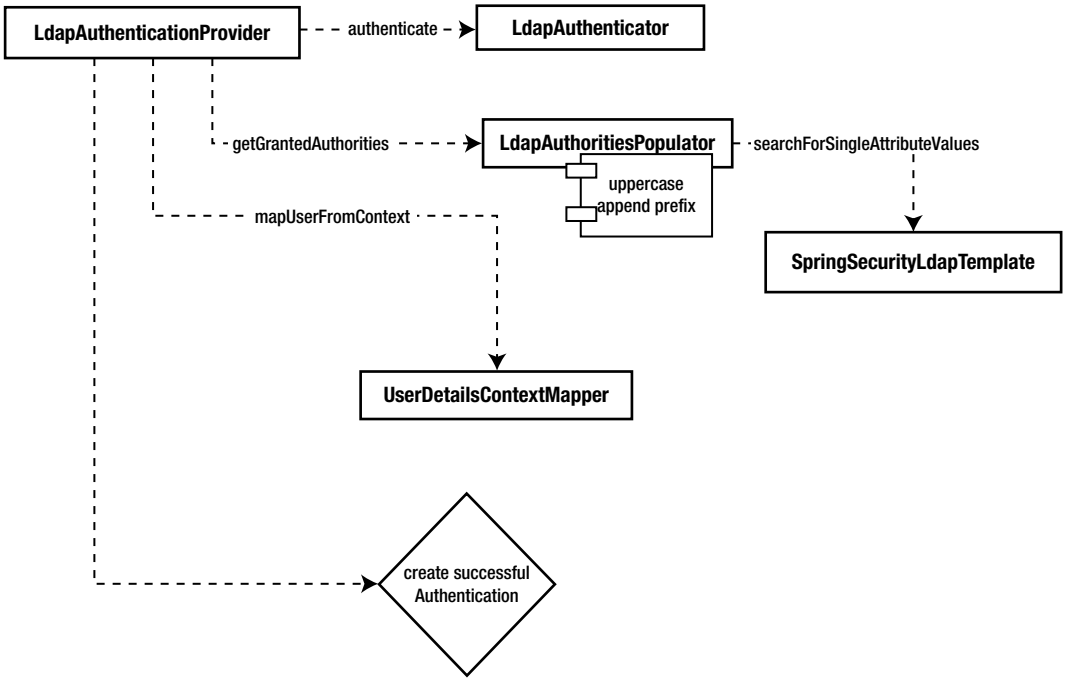


Figure 6-11. *LdapAuthenticationProvider at work*

- When you use any username and password combination (I will use a username of **mon**, and a password of **scarvarez** to illustrate) to log in when accessing the URL `http://localhost:8080/hello`, the request arrives at the `LdapAuthenticationProvider`'s `authenticate` method. This class and method delegate the authentication process to an instance of `LdapAuthenticator`. By default, the implementation used is `BindAuthenticator`, which as I mentioned before authenticates the user using a bind operation against the LDAP server.
- If the user is authenticated successfully by the `LdapAuthenticator`, an `LdapAuthoritiesPopulator` implementation is called. In particular, the `DefaultLdapAuthoritiesPopulator`'s `getGrantedAuthorities method` implementation is called. This class uses an instance of `org.springframework.security.ldap.SpringSecurityLdapTemplate` to query the LDAP directory and retrieve the `cn` attribute of the groups of which the retrieved user is a member. This is done in the method `searchForSingleAttributeValues`. The retrieved values from the `cn` attributes of the groups (which, in this case, are "administrators" and "users") are then converted to uppercase and get prefixed with the default prefix of `ROLE_` before being added to the list of authorities for the logged-in user. This means that, in the case of the user "mon," the authority `ROLE_ADMINISTRATORS` will be returned.
- The next step is that an implementation of `UserDetailsContextMapper` is called in order to create a `UserDetails` object (or, more exactly, one of its implementations) from the LDAP context information that was retrieved from the server and the retrieved authorities. The implementation used by default is `LdapUserDetailsMapper`.
- Then a successful `Authentication` object in the form of a `UsernamePasswordAuthenticationToken` instance is created with the corresponding user details, and the normal standard process continues through the filter chain.

Other Attributes and Elements in the LDAP Spring Security Namespace

We have developed a fully functional Spring Security LDAP authentication mechanism using the Spring Security XML namespace. This namespace offers some more attributes and elements you can use to configure different parts of the integration.

The element `<ldap-authentication-provider>` supports the following main attributes beside the ones we used in our example:

- `group-role-attribute` This attribute is used to determine the name of the attribute in the group entries that will be used to retrieve the role name for the users belonging to the group. By default, the attribute used is `cn` when no attribute at all is specified explicitly, as you saw in the example.
- `role-prefix` Allows you to determine the prefix to be used to append to the roles when they are retrieved from the LDAP server. As you have seen, it defaults to `ROLE_`.
- `user-context-mapper-ref` This attribute allows you to reference a custom `UserDetailsContextMapper` implementation that can be used to map the LDAP context of the user in a different way than a `UserDetails` object.

The element `<ldap-authentication-provider>` also supports a child element: `<password-compare>`. When this element is used, a different kind of authentication process happens in the application. Actually, a whole new implementation of `LdapAuthenticator` is used—namely, a `PasswordComparisonAuthenticator`. The Spring namespace parsing process takes care of instantiating this class instead of the `BindAuthenticator`.

The difference between `PasswordComparisonAuthenticator` and `BindAuthenticator` is that the `PasswordComparisonAuthenticator` is used to compare the provided password at login time with the password stored in the LDAP repository. This is done by performing an LDAP "compare" operation, through the use of an instance of `SpringSecurityLdapTemplate`, between both passwords. The `BindAuthenticator`, as I said before, uses the standard bind functionality in LDAP to authenticate the user.

The `<password-compare>` element accepts a couple of attributes and one child element:

- The first attribute it accepts is `hash`, which allows you to specify one of the following hashing algorithms for passwords: `plaintext`, `sha`, `sha-256`, `md5`, `md4`, `{sha}`, `{ssha}`.
- The second attribute is `password-attribute`, which allows you to specify what attribute in the user entries on the LDAP directory contains the password for the user, which by default is `userPassword`, which we used in the LDIF file.
- The element it accepts is `<password-encoder>`, which again allows you to specify what kind of algorithm encoding mechanism it uses to encrypt the provided password before comparing it with the one stored in the directory.

The `hash` attribute on `<password-compare>` and the element `<password-encoder>` should not be used together at the same time because they refer to the same kind of information. If both are specified, the `hash` attribute will be ignored. Using the `hash` attribute can be thought of a shortcut for the standard digest algorithms, and internally it creates a password encoder. In particular, the ones from Listing 6-8 are supported. That listing is extracted directly from Spring Security's source code for the class `org.springframework.security.config.authentication.PasswordEncoderParser`.

Listing 6-8. Default password encoders that can be configured using the hash attribute on <password-compare>

```
static final String ATT_REF = "ref";
public static final String ATT_HASH = "hash";
static final String ATT_BASE_64 = "base64";
static final String OPT_HASH_PLAINTEXT = "plaintext";
static final String OPT_HASH_SHA = "sha";
```

```
static final String OPT_HASH_SHA256 = "sha-256";
static final String OPT_HASH_MD4 = "md4";
static final String OPT_HASH_MD5 = "md5";
static final String OPT_HASH_LDAP_SHA = "{sha}";
static final String OPT_HASH_LDAP_SSHA = "{ssha}";

private static final Map<String, Class<? extends PasswordEncoder>> ENCODER_CLASSES;

static {
    ENCODER_CLASSES = new HashMap<String, Class<? extends PasswordEncoder>>();
    ENCODER_CLASSES.put(OPT_HASH_PLAINTEXT, PlaintextPasswordEncoder.class);
    ENCODER_CLASSES.put(OPT_HASH_SHA, ShaPasswordEncoder.class);
    ENCODER_CLASSES.put(OPT_HASH_SHA256, ShaPasswordEncoder.class);
    ENCODER_CLASSES.put(OPT_HASH_MD4, Md4PasswordEncoder.class);
    ENCODER_CLASSES.put(OPT_HASH_MD5, Md5PasswordEncoder.class);
    ENCODER_CLASSES.put(OPT_HASH_LDAP_SHA, LdapShaPasswordEncoder.class);
    ENCODER_CLASSES.put(OPT_HASH_LDAP_SSHA, LdapShaPasswordEncoder.class);
}
```

As you can see from the example, configuring LDAP's basic support as the authentication solution for your application with Spring Security is not that complex. In fact, it is very straightforward thanks to the modular architecture and the well-thought-out XML namespace. For me, the complexity with LDAP is LDAP itself. Although it is a simple hierarchical system (very much like the file system in your standard Unix box), some of the nomenclature and functionality seems a bit complex and very different from the database-based solution you explored in the previous section.

As I said before, using LDAP as your authentication solution makes great sense in the context of corporate intranets, where the company user base is already stored in LDAP-like directories in a centralized manner. Plugins into this already existing user-management infrastructure are a good way to reuse the user information within the company instead of writing a parallel authentication datastore that then needs to be kept in sync with the main repository.

Authenticating with OpenID

OpenID is an authentication solution that exists to address the problems inherent in having many user accounts in many different sites with many different sets of credentials.

It is common on the web for a user to be registered with many sites and applications at the same time. Most of these applications, when you try to access them, ask you to authenticate against them directly—normally, asking you for a username and a password to access their functionality. This is an obvious problem in terms of inconvenience and security.

OpenID allows users to rely on unique single authentication account to access different web sites. In this way, it reduces the inconvenience of maintaining multiple accounts on multiple web sites. Also, it reduces the risk of a password being compromised on any external site, because the OpenID identity provider manager is the only one that will know about your password.

There are many OpenID providers, including Google and Yahoo, as well as dedicated ones like myOpenID, which is the one I will use in the example. You can open an account with myOpenID by visiting https://www.myopenid.com/. As I have been doing so far in the book, I will explain the concepts as I am developing the example.

OpenID is an authentication mechanism. It doesn't include any authorization-related functionality, so it will probably need to be complemented with another mechanism to add the authorities and roles to the users.

Once again, we will use the simple application we have been using in the other examples in this chapter, with the required modifications needed to support OpenID authentication.

Setting Up OpenID Authentication

Next, let's configure the simplest possible OpenID-enabled Spring Security application. Listing 6-9 shows the configuration file `applicationContext-security.xml` for this application, and Listing 6-10 shows the new Maven dependencies you need to include in your project's `pom.xml` file. These libraries are the Spring Security OpenID module and the `nekohtml` (available at `http://nekohtml.sourceforge.net/`) library, which is used internally by the `openid4java` library (which is used by Spring Security and is resolved as a transitive dependency) for parsing HTML.

Listing 6-9. Configuration file for a Spring Security application with OpenID enabled

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

    <security:http auto-config="true">
        <security:intercept-url pattern="/*" access="ROLE_ADMINISTRATOR" />
        <security:openid-login/>
    </security:http>

    <security:user-service id="userService">
        <security:user name="http://carlo8172.myopenid.com/" authorities="ROLE_ADMINISTRATOR"/>
    </security:user-service>

    <security:authentication-manager alias="authenticationManager"/>

</beans>
```

Listing 6-10. Maven dependencies for OpenID support

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-openid</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>

<dependency>
    <groupId>net.sourceforge.nekohtml</groupId>
    <artifactId>nekohtml</artifactId>
    <version>1.9.16</version>
</dependency>
```

In Listing 6-9, you can see that I included the element `<security:openid-login/>` as a child of the `<http>` element. This element makes sure that Spring's startup process instantiates for us a new `org.springframework.security.openid.OpenIDAuthenticationFilter` (and adds it to the security filter chain)

and a new `org.springframework.security.openid.OpenIDAuthenticationProvider`, with its corresponding `UserDetailsService` implementation injected into it will be added into the `AuthenticationManager` list of tested providers. The element `<openid-login>` also alters the `org.springframework.security.web.authentication.ui.DefaultLoginPageGeneratingFilter`, which generates the login form, to add a text field to introduce the OpenID identification of the user.

Also in Listing 6-9, you can see that I included a `<user-service>` element that contains a strange username and the authorities (in this case, only the `ROLE_ADMINISTRATOR` authority) but doesn't contain any password information. The password is stored only in the OpenID provider environment (MyOpenID). The strange username of `http://carlo8172.myopenid.com/` is my OpenID account identifier. If you open an account yourself, it will be very similar to this one.

You might find it a bit strange to define the user in the application in the `<user-service>` element if it is already defined in the MyOpenID site. The reason is that, as I said before, you need to specify the authorities that your users will have within the application and you cannot do that in the OpenID provider because it provides only an authentication mechanism. What you could do, however, is autoregister the users after they authenticate successfully with the OpenID provider and assign them a particular user role. For example, many applications use only one generally available user role that is common to all the users that access the application. In this case, you could simply assign that role by default to the user after she has been successfully authenticated by the OpenID provider. This is a common way of working, and the Spring Security source code includes an example that shows exactly this kind of behavior by implementing a custom `UserDetailsService` for use with the LDAP authentication provider. Figure 6-12 shows this idea clearly with an image.

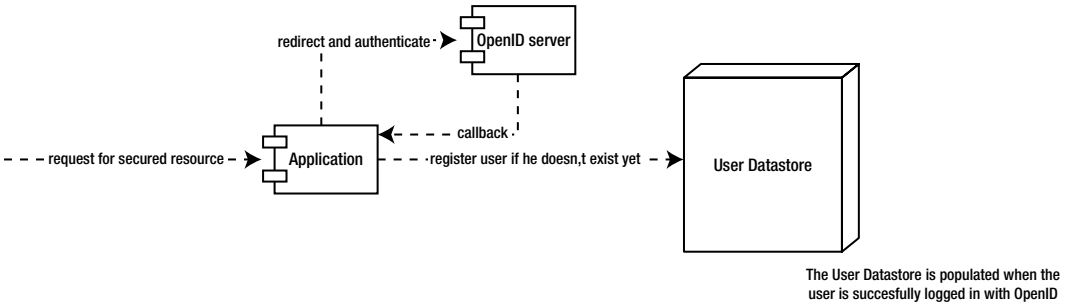


Figure 6-12. *OpenID auto-register functionality*

Now restart the application, and visit the URL `http://localhost:8080/hello`. You should get to a page that looks like Figure 6-13, which presents you with a login form that has a new text field.

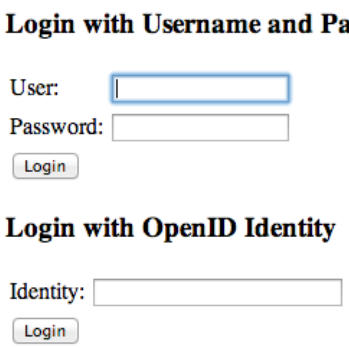


Figure 6-13. *Default login form now includes the option to log in with OpenID*

In the new Identity text field, I type my OpenID identifier **http://carlo8172.myopenid.com/**, and then click the second Login button. I get redirected to the MyOpenID site, which shows me a form to type in my password. This is shown in Figure 6-14.



Figure 6-14. *The MyOpenID site asking me for my password to authenticate me*

After I correctly enter my password in the MyOpenID site, I get redirected back to the application. Now, if I visit the localhost URL `/hello`, I get the familiar "Hello World" message, meaning that I have been correctly authenticated and the correct authorities have been assigned to my user.

OpenID Authentication Flow

The way all this works is similar to the other authentication mechanisms you saw earlier, and many of the known abstractions still apply here, like the `AuthenticationProvider` and `UserDetails` abstraction. Step by step, what happens is the following:

1. When you first visit the localhost URL `/hello`, because it is configured to be accessed only by users with role `ROLE_ADMINISTRATOR` and there is still no logged-in user, you get redirected to the URL `/spring_security_login`. This URL is detected by the `DefaultLoginPageGeneratingFilter`, which as you saw previously, generates the login form page. This time, however, the filter sees that OpenID is enabled in the application and generates one extra form like this:

```
<form name="oidf" action="/j_spring_openid_security_check" method="POST">
```

The form has a text field for entering the OpenID identifier like this:

```
<input type="text" size="30" name="openid_identifier">
```

2. When you enter the OpenID identifier in the text box and click Login, the request arrives at the `OpenIDAuthenticationFilter`, which detects the invoked URL `/j_spring_openid_security_check` as the URL it needs to process. The filter then extracts the parameter `openid_identifier` from the request. With the username it uses the `openid4java` library to discover the OpenID server provider to which the authentication

request belongs. Then it creates an OpenID authentication request with this information and redirects the response to the corresponding OpenID provider site.

3. When you introduce the correct password in the OpenID provider (MyOpenID, in this case), you get redirected back to the URL `/j_spring_openid_security_check` once again. However, this time the URL contains a set of parameters that MyOpenID has included in the invocation. The filter `OpenIDAuthenticationFilter` again recognizes this URL as a URL it should process and proceeds to extract the parameter `opened.identity` from the request. (This parameter was added by the OpenID provider, as I just said.)

4. Then the information sent by the provider is verified to see if it was successful. Again, this is the work of the library `openid4java`.

5. After successful verification, an instance of `OpenIDAuthenticationToken` is created and sent to the `OpenIDAuthenticationProvider`'s `authenticate` method. The provider calls the configured `UserDetailsService` instance (an `InMemoryUserDetailsManager`) and retrieves the user configured for the given username.

6. The `OpenIDAuthenticationProvider` then creates a successful `OpenIDAuthenticationToken Authentication` implementation, which includes the `UserDetails` information, including the authorities retrieved with the `UserDetailsService`. As normal, the `Authentication` gets stored in the `SecurityContext` of the running application.

7. When you visit the URL `/hello` again, the normal process of authorization will start. The authorities for the user match those required by the requested URL, so access will be granted to the URL.

Spring Security OpenID Namespace

Spring Security's OpenID namespace configuration provides other attributes and elements that enhance the functionality you have seen up to this point. In general, though, the `spring-security-openid` module is very small and the core functionality is what I have covered so far in this section. The main supported attributes in the `<openid-login>` configuration element are pretty much the same as the ones for `<form-login>` that you explored before in the book. They are summarized in the following list, with their definitions extracted directly from the available source code—particularly, from the `spring-security-config` module in the file `spring-security-3.1.xsd`:

- `always-use-default-target` Indicates whether the user should always be redirected to the `default-target-url` after login.
- `authentication-failure-handler-ref` Reference to an `AuthenticationFailureHandler` bean that should be used to handle a failed authentication request. It should not be used in combination with `authentication-failure-url`, because the implementation should always deal with navigation to the subsequent destination. This means that the intended work of `authentication-failure-url` doesn't make sense by itself because its job is taken care of by the handler.
- `authentication-failure-url` The URL for the login failure page. If no login failure URL is specified, Spring Security automatically creates a failure login URL at `/spring_security_login?login_error` and a corresponding filter to render that login failure URL when requested.

- `authentication-success-handler-ref` Reference to an `AuthenticationSuccessHandler` bean that should be used to handle a successful authentication request. It should not be used in combination with `default-target-url` (or `always-use-default-target-url`) because the implementation should always deal with navigation to the subsequent destination. This means that the intended work of `default-target-url` doesn't make sense by itself because its job is taken care of by the handler.
- `default-target-url` The URL that will be redirected to after successful authentication if the user's previous action could not be resumed. This generally happens if the user visits a login page without first requesting a secured operation that triggers authentication. If unspecified, it defaults to the root of the application.
- `login-page` The URL for the login page. If no login URL is specified, Spring Security automatically creates a login URL at `/spring_security_login` and a corresponding filter to render that login URL when requested.
- `login-processing-url` The URL that the login form is posted to. If unspecified, it defaults to `/j_spring_security_check`.

The `<openid-login>` element also accepts a child element: `<attribute-exchange>`. This element is used to use the OpenID Attribute Exchange (AX) functionality. The AX functionality of OpenID allows for the interchange of information beyond simple authentication between the communicating endpoints of the authentication process. For example, you could use it to fetch data from the OpenID provider or to store data. I will show you how to configure it to retrieve some extra data when you are using the authentication process. I will configure it to retrieve the email of the logged-in user.

If you are following the example and using MyOpenID as your OpenID provider, go to the URL `https://www.myopenid.com/settings` (when you are logged in already) and, in the right panel, on the Your Account menu, click the Registration Personas item. This allows you to add information to your account profile in the MyOpenID provider. After you add the information you want, it should look something like Figure 6-15.

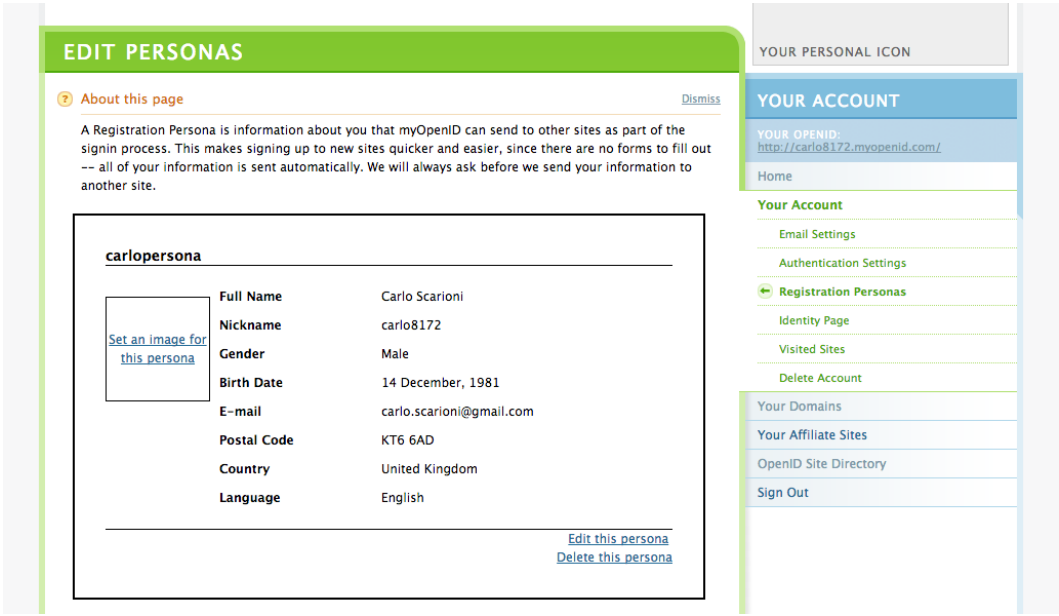


Figure 6-15. *Adding extra information to the account profile in MyOpenID*

After you add this information to your account on the OpenID provider, you can ask for this information back when authenticating against it. For example, let's say that you want to retrieve the email of the user attempting to log in. To do that, you add the configuration from Listing 6-10 as a child of the element `<openid-login>` in your Spring configuration file.

Listing 6-10. Attribute exchange configuration for fetching the email of the user

```
<security:attribute-exchange>
    <security:openid-attribute name="email" type="http://schema.openid.net/contact/email"
required="true"/>
</security:attribute-exchange>
```

Listing 6-10 shows how to retrieve a particular attribute from the OpenID account when logging in. You can see that the `type` attribute specifies a particular URI that identifies how to retrieve this value in the particular OpenID provider. I say in the "particular provider" because although there is some effort to standardize the AX attributes, each provider often uses different ways to refer to the same things. The `name` attribute specifies the name that the attribute has in the local `Authentication` object after it is retrieved.

If you restart the application now and do the whole authentication process again, this time your email will be returned with a successful authentication with the OpenID provider. The information will be stored in the `OpenIDAuthenticationToken Authentication` object under the `attributes` property, and you can access this information any time you want from the `Authentication` object that will be stored in the `SecurityContext`.

As I said before, there are many other OpenID providers out there, including Google and Yahoo, and they work in a similar manner to what you saw in this section with MyOpenID. The main difference, in many cases, is that the user identifier URI related to each of the providers.

X.509 Authentication

X.509 authentication is an authentication scheme that uses client-side certificates instead of username-password combinations to identify the user. Using this approach, a scheme known as *mutual authentication* takes place between the client and the server. In practice, mutual authentication means that, as part of the Secure Sockets Layer (SSL) handshake, the server requests that the client identify himself by providing a certificate. In a production-ready server, the incoming client certificate needs to be issued and signed by a proper certificate-signing authority.

To work with client certificates, the application needs to be configured to use SSL channels in the sections that are expected to deal with the authenticated user, because the X.509 authentication protocol itself is part of the SSL protocol. In Chapter 5, you configured the example application to use SSL channel and configured the `pom.xml` to be able to run a Jetty server with SSL support. Here, I build on that example's `pom.xml` file to explain the client-certificate solution. You need to add a couple of properties to the Jetty plugin configuration. Listing 6-11 shows this part of the `pom.xml` file. Look at the properties `wantClientAuth` and `needClientAuth` that are configured in the last part of the plugin configuration.

Listing 6-11. Jetty Maven configuration in the pom.xml file

```
<plugin>
   <groupId>org.mortbay.jetty</groupId>
   <artifactId>jetty-maven-plugin</artifactId>
   <version>8.1.1.v20120215</version>
   <configuration>
               <connectors>
                  <connector implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
                     <port>8080</port>
                     <maxIdleTime>60000</maxIdleTime>
                  </connector>
```


Download from Wow! eBook <www.wowebook.com>

```
                <connector implementation="org.eclipse.jetty.server.ssl.SslSocketConnector">
                   <port>8443</port>
                   <maxIdleTime>60000</maxIdleTime>
                   <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
                   <password>jetty8</password>
                   <keyPassword>jetty8</keyPassword>
                   <wantClientAuth>true</wantClientAuth>
                   <needClientAuth>true</needClientAuth>
                </connector>
             </connectors>
   </configuration>
</plugin>
```

The first thing you do now is to enable X.509 authentication on the Spring configuration file. Listing 6-12 shows the configuration file `applicationContext-security.xml`.

Listing 6-12. applicationContext-security.xml configured with X.509 client certificate authentication

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

 <security:http>
    <security:x509 subject-principal-regex="CN=(.*?),"/>
        <security:intercept-url pattern="/*"
            access="ROLE_ADMINISTRATOR" requires-channel="https" />
 </security:http>

    <security:authentication-manager alias="authenticationManager">
        <security:authentication-provider>
            <security:user-service>
                <security:user name="car" authorities="ROLE_ADMINISTRATOR" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>

</beans>
```

Listing 6-12 essentially has only one new element: `<security:x509>`. This element is all Spring Security needs in order to do its part of the work in activating the client-certificate authentication mechanism. By configuring this element, Spring's startup mechanism makes sure to instantiate an instance of `org.springframework.security.web.authentication.preauth.x509.X509AuthenticationFilter` and add it to the Spring Security filter chain. This is all the configuration Spring needs to handle client certificates.

Of course, things are not that simple and the job is not yet done. In reality, Spring Security X.509 support doesn't authenticate the user. The user is assumed to be already authenticated, and Spring Security simply creates a successful `Authentication` object with information extracted from the certificate and stores it in the `SecurityContext` in the standard way.

So the entity that is actually in charge of authenticating the user (or more exactly accepting the user as a properly identified one) is the web server, which does this by accepting the provided client certificate. Basically, if the server decides that the certificate sent by the user is a valid one, the user is who she claims to be and gets authenticated.

In a production system, the web server makes sure that the certificate provided by the client is signed by an authorized trusted authority. However, we will use a test environment with our common Maven-configured Jetty installation, and for that we will configure a self-signed certificate and make sure that Jetty accepts it.

Let's create the certificate. It is pretty straightforward to do, and we will use the `openssl` tool to do it:

1. Generate a private key by executing the following command:

   ```
   openssl genrsa -out private.key 1024
   ```

Figure 6-16 shows this first command.

```
/tmp/tests $ openssl genrsa -out private.key 1024
Generating RSA private key, 1024 bit long modulus
.....................++++++
.++++++
e is 65537 (0x10001)
```

Figure 6-16. *Generating a private key*

2. Generate a certificate by executing the following command and pressing Enter at every prompt:

   ```
   openssl req -new -x509 -days 365 -key private.key -out certificate.crt
   ```

Figure 6-17 shows the output from this command.

```
/tmp/tests $ openssl req -new -x509 -days 365 -key private.key -out certificate.crt
You are about to be asked to enter information that will be incorporated
into your certificate request.
What you are about to enter is what is called a Distinguished Name or a DN.
There are quite a few fields but you can leave some blank
For some fields there will be a default value,
If you enter '.', the field will be left blank.
-----
Country Name (2 letter code) [AU]:
State or Province Name (full name) [Some-State]:
Locality Name (eg, city) []:
Organization Name (eg, company) [Internet Widgits Pty Ltd]:
Organizational Unit Name (eg, section) []:
Common Name (e.g. server FQDN or YOUR name) []:
Email Address []:
/tmp/tests $ _
```

Figure 6-17. *Generating a certificate*

When executing that last step, it is important that when asked to introduce your name by the command-line prompt, you use the name "car" because that is the username you configured in the application configuration file. The command line will show you the following prompt *Common Name (e.g. server FQDN or YOUR name) []:car*.

That's it! That process created a self-signed X509 certificate with 365 days of duration signed with the private key.

The next thing you need to do is to make Jetty trust the certificate authority associated with this certificate. To do that, you use the `keytool` tool to create a new trust store with the generated certificate from before. You need to use a password when prompted. I used *changeit*.

```
keytool -importcert -alias somealias -keystore jetty-ssl.truststore -file certificate.crt
```

■ **Note** A truststore is a repository of certificates that are trusted by the JRE that uses such a truststore. By default the JRE truststore trusts any certificates signed by a recognised Certificate Authority (CA). If you have a certificate not signed by a CA (like the self signed certificates) you will need to add it manually to the truststore so the it can be trusted.

The execution of that file is shown in Figure 6-18. That simple line created the file `jetty-ssl.truststore`, which you will use as the trust store for the Jetty server to accept the user certificate. Copy that file to the root of the application, and modify the `pom.xml jetty plugin` section to include the `line <truststore>jetty-ssl.truststore</truststore>` as part of the `org.eclipse.jetty.server.ssl.SslSocketConnector` configuration.

```
/tmp/tests $ keytool -importcert -alias somealias -keystore jetty-ssl.truststore -file  certificate.crt
Enter keystore password:
Re-enter new password:
Owner: O=Internet Widgits Pty Ltd, ST=Some-State, C=AU
Issuer: O=Internet Widgits Pty Ltd, ST=Some-State, C=AU
Serial number: 84a400bd6d8886fe
Valid from: Sun Dec 30 18:02:22 GMT 2012 until: Mon Dec 30 18:02:22 GMT 2013
Certificate fingerprints:
         MD5:  0C:CA:0B:CD:56:B4:A1:5A:09:B2:64:3C:04:C1:2D:4C
         SHA1: DB:30:4D:F7:4D:DB:68:92:72:98:86:2A:5E:B6:BD:3F:E8:04:D0:3E
         SHA256: B3:0C:5D:31:78:37:8C:52:5A:95:AD:62:08:3F:64:34:B6:E0:CB:71:12:8F:AA:E9:5C:C3:7E:6F:33:2E:DA:2C
         Signature algorithm name: SHA1withRSA
         Version: 3

Extensions:

#1: ObjectId: 2.5.29.14 Criticality=false
SubjectKeyIdentifier [
KeyIdentifier [
0000: 9A EE 21 E0 99 51 6A 30   0F D6 E6 C8 85 86 28 F7  ..!..Qj0......(.
0010: 15 19 62 4C                                        ..bL
]
]

#2: ObjectId: 2.5.29.35 Criticality=false
AuthorityKeyIdentifier [
KeyIdentifier [
0000: 9A EE 21 E0 99 51 6A 30   0F D6 E6 C8 85 86 28 F7  ..!..Qj0......(.
0010: 15 19 62 4C                                        ..bL
]
]

#3: ObjectId: 2.5.29.19 Criticality=false
BasicConstraints:[
  CA:true
  PathLen:2147483647
]

Trust this certificate? [no]:  yes
Certificate was added to keystore
/tmp/tests $
```

Figure 6-18. *A truststore created with an imported certificate*

Next you need to add the certificate to the browser you will use to connect to the application. You will use Firefox in this example. The first thing to do is create a p12 file that contains information about the certificate and the private key associated with the certificate. This is the file needed by the browser to identify the user. To generate that file from the command line, in the same directory where you previously generated the key and the certificate, execute the following command.

```
openssl pkcs12 -export -in certificate.crt -inkey private.key > client.p12
```

The result can be seen in Figure 6-19. You use no password when prompted.

```
/tmp/tests $ openssl pkcs12 -export -in certificate.crt -inkey private.key >  client.p12
Enter Export Password:
Verifying - Enter Export Password:
/tmp/tests $ ls
certificate.crt  client.p12  jetty-ssl.truststore  private.key
/tmp/tests $
```

Figure 6-19. *Generating the certificate pkcs fot importing in the browser*

To import that file into the client certificates of Firefox, do the following. (This example is on a Mac computer, and it should be similar on other systems.)

1. Click Firefox ➤ Preferences.
2. Click the Advanced tab.
3. Click the Encryption tab.
4. Click View Certificates.
5. Click Import.
6. Locate client.p12 in the directory where it is stored, and double-click it.
7. Leave the password prompt empty.

That's it. The certificate is imported, and you should see a screen like Figure 6-20.

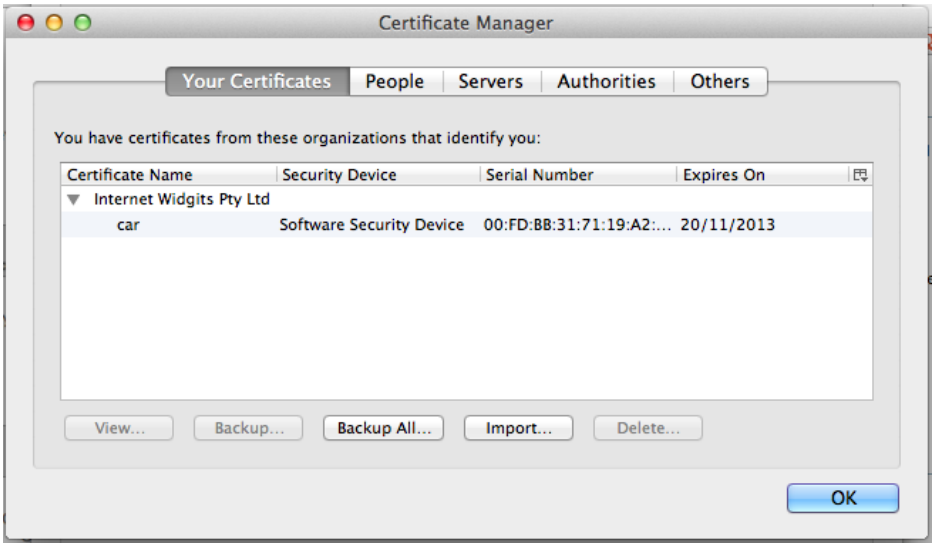


Figure 6-20. *The client certificate has been successfully imported into Firefox*

Finally, the configuration is complete, and the application is ready. The browser is also ready for the mutual authentication process to get underway. To try it out, restart the server:

```
mvn clean install jetty:run
```

Then visit the URL `/hello`. You need to accept the self-signed certificate from the server to continue. After you accept it, you will be presented with the screen shown in Figure 6-21, which asks you to select the certificate to use to authenticate. Only one option is available: the certificate you just imported. After clicking OK, you get to the familiar "Hello World" page, meaning that the authentication (and the subsequent authorization with the configured `UserDetailsService`) was successful.

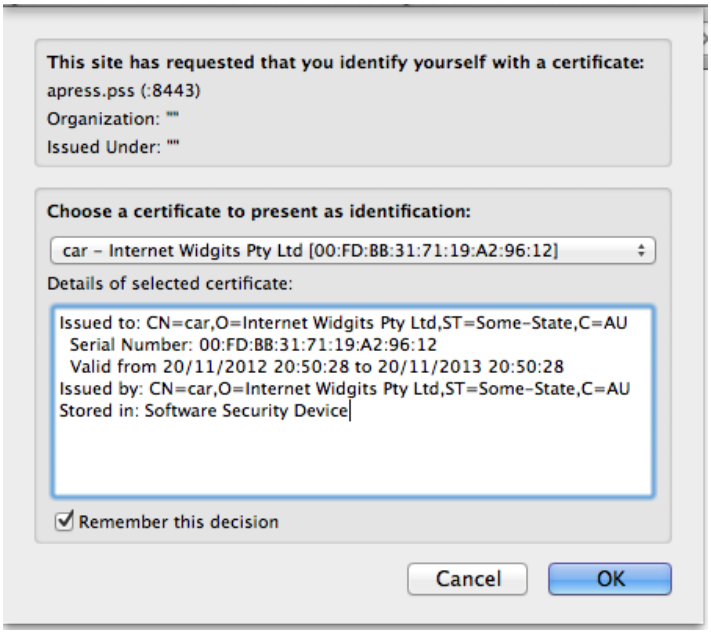


Figure 6-21. *Selecting a client certificate to use*

This is the way the process works:

1. Most of the process is dealt with by the web browser and the web server SSL communication establishing the mutual authentication scheme.
2. Once the browser sends the client certificate, the request arrives at Spring Security's filter chain—in particular, at the configured `X509AuthenticationFilter`, which is a subclass of `AbstractPreAuthenticatedProcessingFilter`. `AbstractPreAuthenticatedProcessingFilter` is a base filter implementation that handles the special cases where a request arrives with information about a user already authenticated by an external system. Filters extending this class assume that the user is already authenticated and that their only responsibility is to extract the authentication details from the request, create an instance of `PreAuthenticatedAuthenticationToken`, and pass it forward to the `AuthenticationManager` implementation.
3. `X509AuthenticationFilter` looks for the attribute `javax.security.cert.X509Certificate` in the request. There, it finds the whole client certificate sent by the browser in the request.

4. With the extracted certificate, the filter calls an instance of an implementation of org.springframework.security.web.authentication.preauth.x509.X509PrincipalExtractor to retrieve the username from the said certificate. Here the implementation (`SubjectDnX509PrincipalExtractor`) finds the username "car" in the certificate, extracting it from the Common Name (`cn`) you configured when you first created the certificate. This is why it was important to use the name "car" when asked in the command prompt. Looking at the <x509> element on the `applicationContext-security.xml configuration` file, you can see that the attribute subject-principal-regex="CN=(.*?)" is configured. This regex pattern is what is used by the SubjectDnX509PrincipalExtractor to find the username in the certificate.

5. The Authentication object implementation, PreAuthenticatedAuthenticationToken, is instantiated with the username and the credentials. The credentials that will be stored in the Authentication object are the full sun.security.x509.X509CertImpl instance that was extracted from the request.

6. The configured AuthenticationManager is called, which in turn calls the AuthenticationProvider implementation PreAuthenticatedAuthenticationProvider. This provider uses the configured UserDetailsService to retrieve the user for the application using the username from the `Authentication` object. In the example, it uses the in-memory UserDetailsService you defined in the application file.

7. After it retrieves the UserDetails from the UserDetailsService, the provider creates a new fully authenticated PreAuthenticatedAuthenticationToken that now includes the authorities granted to the user as extracted from the UserDetailsService.

That is the whole flow that is followed by a request using client-certificate authentication. As you can see, Spring Security support for it is pretty straightforward.

JAAS Authentication

The Java Authentication and Authorization Service (JAAS) is the existing standard Java support for managing authentication and authorization. Its functionality clearly overlaps with that of Spring Security.

You can use JAAS as another authentication provider in Spring Security. The JAAS authentication model works in the following way:

1. You initiate the authentication process by creating a new javax.security.auth.login.LoginContext.

2. The instance of LoginContext calls a configured instance of an implementation of javax.security.auth.spi.LoginModule, which is the one that performs the real authentication process.

3. The LoginModule extracts the username and password from the user (or uses a different kind of providing mechanism), and then verifies these credentials.

4. LoginModule communicates with the requesting client by means of callback methods registered with a javax.security.auth.callback.CallbackHandler implementation.

Those are the main steps for authentication with JAAS, and Spring Security can take care of this process with the use of the supplied JAAS authentication providers and some other helper classes. You need to implement a simple `LoginModule` that will take care of the actual authentication of the user. You will use a `map` as your user and password storage.

Next let's create a typical example. I will show you how to modify the code from the previous sections to adapt it to the JAAS authentication mechanism. Again, remember to use the bootstrap application. No specific dependencies are needed because JAAS support in Spring Security is in the core module.

First, let's deal with the Spring configuration. Listing 6-13 shows the configuration file (`applicationContext-security.xml`) needed to use JAAS authentication in Spring Security in our example application.

Listing 6-13. Spring Security configuration file with the JAAS-related configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

   <security:http auto-config="true">
        <security:intercept-url pattern="/*"
            access="ROLE_ADMINISTRATOR" />
    </security:http>

    <security:authentication-manager>
        <security:authentication-provider
            ref="jaasAuthProvider" />
    </security:authentication-manager>

    <bean id="jaasAuthProvider"
        class="org.springframework.security.authentication.jaas.JaasAuthenticationProvider">
        <property name="loginConfig" value="classpath:pss_jaas.config" />
        <property name="authorityGranters">
            <list>
                <bean class="com.apress.pss.security.RoleGranterFromMap" />
            </list>
        </property>
        <property name="loginContextName" value="Pss" />
        <property name="callbackHandlers">
            <list>
                <bean

                    class="org.springframework.security.authentication.jaas.JaasNameCallbackHandler" />
                <bean

                    class="org.springframework.security.authentication.jaas.JaasPasswordCallbackHandler" />
            </list>
        </property>
    </bean>

</beans>
```

Listing 6-13 is not as simple as the files you used in previous sections. For a start, you are using standard Spring bean definitions instead of support from the XML namespace, because there is no XML schema for JAAS configuration. However, it is not that complicated either. The only high-level bean you need to define is for the authentication provider implementation you want to use. The properties you defined in this bean are all important, and here is what they specify:

- `loginConfig` This property specifies the resource location of the file where the definition of the JAAS `LoginModule` can be found. In this case, you are specifying it can be found in the root of the classpath. JAAS configuration files are standard ways of defining the classes that contain the `LoginModules` for your application. The content of file `pss_jaas.config` can be seen in Listing 6-14. I explain later the contents of the class referenced by the file.
- `authorityGranters` This property is used to specify a list of beans capable of adding authorities to users. Implementations of this class ideally query some datastore where the roles for a particular username exist. In the example, you use a simple in-memory map in the class itself as Listing 6-15 shows.
- `loginContextName` Specifies the name of the `LoginModule` that will be used in this particular provider. This name, by default, is `SPRINGSECURITY`. You changed it to match the value defined in the `pss_jaas.config` file.
- `callbackHandlers` Specifies the callback handlers that will take care of processing the standard callback instances created for retrieving the username and password from the user. This method is invoked (as normal) in the `initialize` method of the `LoginModule`. This part is actually a bit tricky to understand, and I'll provide a better explanation when I get to the code of the `SampleLoginModule` class, which is shown in Listing 6-16.

That is all the configuration needed. And all the extra files you need to create are the three shown in Listing 6-14 through Listing 6-16.

Listing 6-14. The pss_jaas.config file that specifies the class implementing LoginModule in the application

```
Pss {
  com.apress.pss.security.SampleLoginModule required;
};
```

Listing 6-15. RoleGranterFromMap, which contains a map with usernames and their assigned roles

```
package com.apress.pss.security;

import java.security.Principal;
import java.util.Collections;
import java.util.HashMap;
import java.util.Map;
import java.util.Set;

import org.springframework.security.authentication.jaas.AuthorityGranter;

public class RoleGranterFromMap implements AuthorityGranter {

    private static Map<String, String> USER_ROLES = new HashMap<String, String>();
```

```
    static {
        USER_ROLES.put("car", "ROLE_ADMINISTRATOR");
    }

    public Set<String> grant(Principal principal) {
        return Collections.singleton(USER_ROLES.get(principal.getName()));
    }
}
```

Listing 6-16. SampleLoginModule takes care of the login process

```
package com.apress.pss.security;

import java.io.Serializable;
import java.security.Principal;
import java.util.HashMap;
import java.util.Map;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;

public class SampleLoginModule implements LoginModule {

    private Subject subject;
    private String password;
    private String username;
    private static Map<String, String> USER_PASSWORDS = new HashMap<String, String>();

    static {
        USER_PASSWORDS.put("car", "scarvarez");
    }

    public boolean abort() throws LoginException {
        return true;
    }

    public boolean commit() throws LoginException {
        return true;
    }

    public void initialize(Subject subject, CallbackHandler callbackHandler,
            Map<String, ?> sharedState, Map<String, ?> options) {
        this.subject = subject;
```

```
        try {
            NameCallback nameCallback = new NameCallback("prompt");
            PasswordCallback passwordCallback = new PasswordCallback("prompt",
                    false);

            callbackHandler.handle(new Callback[] { nameCallback,
                    passwordCallback });

            this.password = new String(passwordCallback.getPassword());
            this.username = nameCallback.getName();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    public boolean login() throws LoginException {

        if (USER_PASSWORDS.get(username) == null
                || !USER_PASSWORDS.get(username).equals(password)) {
            throw new LoginException("username is not equal to password");
        }

        subject.getPrincipals().add(new CustomPrincipal(username));
        return true;
    }

    public boolean logout() throws LoginException {
        return true;
    }

    private static class CustomPrincipal implements Principal, Serializable {
        private final String username;

        public CustomPrincipal(String username) {
            this.username = username;
        }

        public String getName() {
            return username;
        }
    }

}
```

If you restart the application now with all this configuration in place and try to access the URL http://localhost:8080/hello, you get the form to log in that you have seen many times before. If you log in with the username **car** and the password **scarvarez**, you should be able to access the "Hello World" page as in the previous examples.

The preceding code is very simple, and all it's doing is simulating a user data store with an in-memory map. Then it assigns the role `ROLE_ADMINISTRATOR` to a user "car." The only method that it is implementing from the interface `grant` is in charge of returning a `java.util.Set` of authorities given a username.

Download from Wow! eBook <www.wowebook.com>

Listing 6-16 shows the main class you need to create to support JAAS authentication with Spring Security. It is the main class in the JAAS authentication scheme because it is the one that takes charge of the real authentication of the users into the application. People or companies that want to implement a custom authentication solution to be plugged into JAAS definitely need to create an implementation of `LoginModule`.

In our implementation, you can see the use of some of the concepts I was talking about in previous sections, like the `Callbacks` and `CallbackHandler` implementation classes. `NameCallback` and `PasswordCallback` are standard classes in JAAS that will eventually be called back with the contents of username and password, respectively. The `CallbackHandler` implementation is in charge of retrieving the credentials needed to authenticate the user. It then makes sure the relevant callbacks are called in order to populate the needed information for the rest of the process.

After the `CallbackHandler` returns, the `Callback` classes will have the required values set in, and you can extract them and store them for later.

When the login method is called, the introduced user details are easily accessible in the instance variables. Now, they need to be verified. In the example, this simply means making sure that the pair username-password exists in the in-memory map of users defined in the static variable `USER_PASSWORDS` in the class. If a match is not found, a `LoginException` is thrown indicating the authentication failure condition. If a match is found, the `principal` is stored and the execution continues normally.

When you execute the application and try to log in with the username **car** and the password **scarvarez**, the following is what happens inside `JaasAuthenticationProvider`:

- The `Authentication` object populated with the username and password arrives at the `authenticate` method.
- The provider creates a new `javax.security.auth.login.LoginContext` with a constructor parameter that is a `javax.security.auth.callback.CallbackHandler` implementation that delegates handling to instances of `org.springframework.security.authentication.jaas.JaasAuthenticationCallbackHandler`.
- The login method in the `javax.security.auth.login.LoginContext` object created in the previous step is called. This method calls the `initialize` method in your `SampleLoginModule`, passing the `CallbackHandler` created in the previous step. Inside your `initialize` method, a new `NameCallback` and `PasswordCallback` are created and passed to the handle method of the `CallbackHandler`.
- `CallbackHandler` delegates to an `org.springframework.security.authentication.jaas.JaasNameCallbackHandler` for the handling of `NameCallback` and an `org.springframework.security.authentication.jaas.JaasPasswordCallbackHandler` for the handling of `PasswordCallback`. `JaasNameCallbackHandler` and `JaasPasswordCallbackHandler`'s `handle` method are called, with each callback and with the `Authentication` object.
- `JaasNameCallbackHandler` sets the name of the user in the `NameCallback`. The name of the user is obtained from the `Authentication` object.
- The `JaasPasswordCallbackHandler` sets the password of the user in `PasswordCallback`. The password of the user is obtained from the `Authentication` object.
- The `initialize` method in the `SampleLoginModule` extracts the username and password from their respective callbacks and assigns them to instance variables.
- The login method in the `SampleLoginModule` is called. This method compares the username and password obtained in the instance variables in the previous step against the users stored in the datastore (in our example, in the `USER_PASSWORDS` map). If they match, it is assumed to be a successful login to this point. If they don't match, an exception is thrown.
- For the successfully authenticated user, the authorities are extracted with the use of the configured `AuthorityGranter`, which in our example is the custom class `RoleGranterFromMap`. This class returns the roles for the logged-in username.

- A new instance of `org.springframework.security.authentication.jaas.JaasAuthenticationToken` is created as the new fully authenticated `Authentication` object. This new `Authentication` object contains the username and authorities, and it is a valid fully authenticated instance. This instance is returned to the caller and will be used in the rest of the authorization process.

JAAS is a relatively large standard that involves a lot more than the small amount of information I covered here. However, the main concepts are the ones I showed you, and the goal of the section is to show the building blocks for integrating it with Spring Security.

Central Authentication Service (CAS) Authentication

As you can see from the Spring Security source code I presented a few times already, there is a module dedicated to CAS authentication. CAS is an enterprise single sign-on solution built in Java and open source. It has a great supporting community and integrates into many Java projects. CAS provides a centralized place for authentication and access control in an enterprise.

Understanding the ideas behind CAS (and indeed any other single sign-on solution) is not difficult. Although I'm not going to explore CAS in detail, there are a few main concepts you need to know about the CAS system to understand better how the upcoming example works. The major concepts are the following:

- **The CAS server application** This is the centralized web application you deploy, and it's the application that all the client applications you want to use in the single sign-on scheme authenticate against. This application offers certain APIs for handling requests from clients.
- **Services** In CAS, every client application is known as a *service*. All the applications that want to use the single sign-on functionality exist as services from the point of view of the CAS server. Normally, applications provide a callback URL when communicating with the CAS server so that the server can call back into the client application. The URL is normally referred to as the *service URL*.
- **Service tickets** A service ticket in CAS is a random string (that starts with ST-) that is passed from the CAS server to the client application in the callback after a successful login in the CAS server. The ticket serves as an identifier that the client application uses for validating the service against CAS and to get the "success" login message back if that is the case. Service tickets are good for only one ticket-validation attempt.

Those three elements illustrate the concepts from the following examples and explanations.

One important characteristic of CAS is that it is designed to serve as a proxy to different authentication storage solutions. This means that it can be used in combination with LDAP, JDBC, or a few other user stores that contain the real user data. This looks a lot like the way Spring Security leverages these same user data stores.

In this section, you will use CAS to authenticate your users in applications. As of this writing, the current version of CAS is 3.5.1, and that is the one you will use here. The first thing you will do is download the latest version from its home at `http://www.jasig.org/`. The link to download it is `http://downloads.jasig.org/cas/cas-server-3.5.1-release.tar.gz`.

After you download it and uncompress the file, you will find a folder inside the main folder named *modules*. In this folder, you will find a War file with the name `cas-server-webapp-3.5.1.war` if you are using the same version I am using. This is the CAS web server application. Now you will deploy it to a Servlet container. You will be using Jetty here again and to make things simple, you will use Jetty from Maven the same way you have been doing all along. However, this time you will run the war file that is provided by CAS. To do this, create the `pom.xml` file from Listing 6-17 in a directory of your choosing. Remember to replace the path to the war file with the one that applies to your installation.

Listing 6-17. A pom.xml file used to run the CAS war application

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.pss</groupId>
    <artifactId>run-cas</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>keytool-maven-plugin</artifactId>
                <executions>
                    <execution>
                        <phase>generate-resources</phase>
                        <id>clean</id>
                        <goals>
                            <goal>clean</goal>
                        </goals>
                    </execution>
                    <execution>
                        <phase>generate-resources</phase>
                        <id>genkey</id>
                        <goals>
                            <goal>genkey</goal>
                        </goals>
                    </execution>
                </executions>
                <configuration>
                    <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
                    <dname>cn=localhost</dname>
                    <keypass>jetty8</keypass>
                    <storepass>jetty8</storepass>
                    <alias>localhost</alias>
                    <keyalg>RSA</keyalg>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>7.1.6.v20100715</version>
                <configuration>
                    <connectors>
                        <connector implementation="org.eclipse.jetty.server.nio.
                        SelectChannelConnector">
                            <port>9080</port>
                            <maxIdleTime>60000</maxIdleTime>
                        </connector>
```

```
                    <connector implementation="org.eclipse.jetty.server.ssl.SslSocketConnector">
                        <port>9443</port>
                        <maxIdleTime>60000</maxIdleTime>
                        <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
                        <password>jetty8</password>
                        <keyPassword>jetty8</keyPassword>
                    </connector>
                </connectors>
                <scanIntervalSeconds>10</scanIntervalSeconds>
                <stopKey>foo</stopKey>
                <stopPort>9999</stopPort>
                <contextPath>/</contextPath>
                <webApp>
                    /home/cscarioni/programs/cas-server-3.5.1/modules/cas-server-webapp-3.5.1.war
                </webApp>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

■ **Note** Single sign-on is a scheme that allows related applications to share the same user login session. A user authenticates once in the central single sign-on authentication provider and as long as she remains authenticated on the system, she can log in to all the other applications without being asked to provide her credentials again. Also, the other applications don't need to know the user password. Only the CAS server needs to know it.

If you now execute `mvn keytool:genkey` followed by `mvn jetty:deploy-war` from the directory where you have this new `pom.xml` file, the default demo application of the CAS server will execute.

This war file demo application allows users to log in by matching their username and password. For example, you can log in with the username **car** and the password **car**. Go ahead and try it. Visit the URL `https://localhost:9443/login`, and you should get to a page like Figure 6-22.

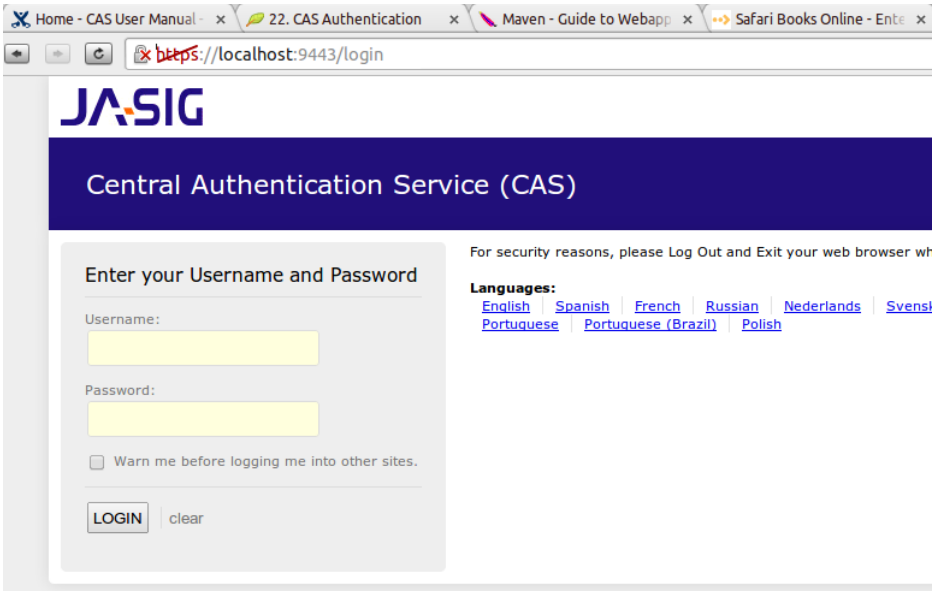


Figure 6-22. *CAS demo application login page*

Log in with the username **car** and the password **car**, and you should get a successful login as shown in Figure 6-23.

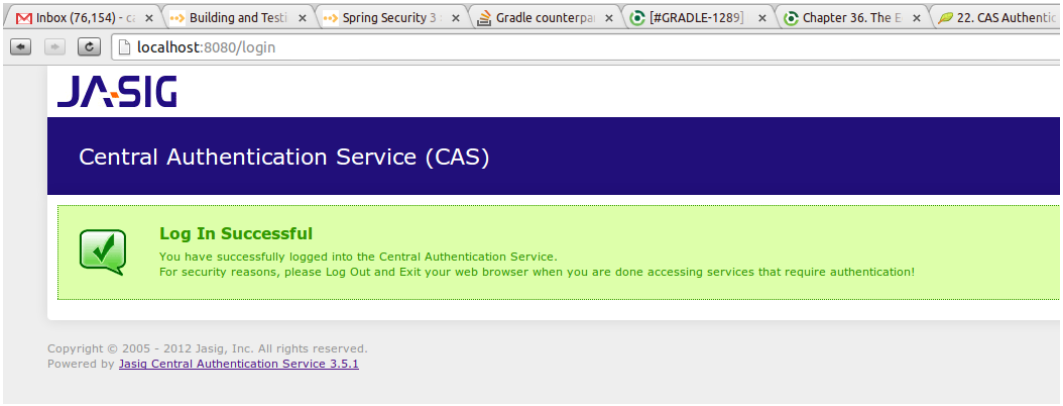


Figure 6-23. *CAS demo application successful login*

Let's create an application now that supports CAS authentication with Spring Security. First let's set up the application , and then I will explain how everything works:

1. From the command line, execute the familiar Maven command to create a new web application:

```
mvn archetype:generate -DgroupId=com.apress.pss.cas
      -DartifactId=uthentication-cas -DarchetypeArtifactId=maven-archetype-webapp
```

2. Replace the generated `pom.xml` file with the one shown in Listing 6-18. This one contains the proper dependencies and the proper Jetty plugin configuration to run the application.

Listing 6-18. The pom.xml file for the CAS authentication-powered application

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.pss.cas</groupId>
    <artifactId>authentication-cas</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>authentication-cas Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>3.0.1</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.5</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-core</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-cas</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-ldap</artifactId>
```

```
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-openid</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>net.sourceforge.nekohtml</groupId>
            <artifactId>nekohtml</artifactId>
            <version>1.9.16</version>
        </dependency>

        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.3</version>
        </dependency>
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>2.2.8</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>authentication-cas</finalName>
        <plugins>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>keytool-maven-plugin</artifactId>
                <configuration>
                    <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
                    <dname>cn=localhost</dname>
                    <keypass>jetty8</keypass>
                    <storepass>jetty8</storepass>
                    <alias>localhost</alias>
                    <keyalg>RSA</keyalg>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>8.1.1.v20120215</version>
                <configuration>
                    <connectors>
                        <connector implementation="org.eclipse.jetty.server.nio.
                        SelectChannelConnector">
```

```
                        <port>8080</port>
                        <maxIdleTime>60000</maxIdleTime>
                    </connector>
                    <connector implementation="org.eclipse.jetty.server.ssl.
                    SslSocketConnector">
                        <port>8443</port>
                        <maxIdleTime>60000</maxIdleTime>
                        <keystore>${project.build.directory}/jetty-ssl.keystore</keystore>
                        <password>jetty8</password>
                        <keyPassword>jetty8</keyPassword>
                    </connector>
                </connectors>
                <scanIntervalSeconds>10</scanIntervalSeconds>
                <stopKey>foo</stopKey>
                <stopPort>8999</stopPort>
                <contextPath>/</contextPath>
            </configuration>
        </plugin>
    </plugins>
</build>
</project>
```

3. Create the `web.xml` file from Listing 6-19 in the WEB-INF directory of the created application. This file, as we have seen many times, has the listener to set up the Spring context.

Listing 6-19. The web.xml file for the CAS-secured application

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

    <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
           /WEB-INF/applicationContext-security.xml
        </param-value>
    </context-param>

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

4. Create the servlet shown in Listing 6-20 in the package `com.apress.pss.servlets`. This is a simple servlet that you will secure next.

Listing 6-20. Simple "Hello World" servlet that you will secure

```
package com.apress.pss.servlets;

import java.io.IOException;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns={"/hello"})
public class HelloWorldServlet extends HttpServlet {
    private static final long serialVersionUID = 2218168052197231866L;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response){
        try {
            response.getWriter().write("Hello World");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

5. Create the file `applicationContext-security.xml` shown in Listing 6-21 in the WEB-INF folder. This file contains the configuration for CAS security that I will explain next.

Listing 6-21. The applicationContext-security.xml spring file with CAS configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:jdbc="http://www.springframework.org/schema/jdbc"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/jdbc
    http://www.springframework.org/schema/jdbc/spring-jdbc-3.1.xsd">

    <security:http auto-config="true" entry-point-ref="casEntryPoint">
        <security:intercept-url pattern="/*" access="ROLE_USER" />
        <security:custom-filter position="CAS_FILTER"
            ref="casFilter" />
    </security:http>
```

```
    <security:user-service id="userService">
        <security:user name="car" authorities="ROLE_USER" />
    </security:user-service>

    <security:authentication-manager alias="authenticationManager">
        <security:authentication-provider ref="casAuthenticationProvider" />
    </security:authentication-manager>

    <bean id="serviceProperties" class="org.springframework.security.cas.ServiceProperties">
        <property name="service"
            value="https://localhost:8443/j_spring_cas_security_check" />
        <property name="sendRenew" value="false" />
    </bean>

    <bean id="casFilter"
        class="org.springframework.security.cas.web.CasAuthenticationFilter">
        <property name="authenticationManager" ref="authenticationManager" />
    </bean>

    <bean id="casEntryPoint"
        class="org.springframework.security.cas.web.CasAuthenticationEntryPoint">
        <property name="loginUrl" value="https://localhost:9443/login" />
        <property name="serviceProperties" ref="serviceProperties" />
    </bean>

    <bean id="casAuthenticationProvider"
        class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
        <property name="authenticationUserDetailsService">
            <bean  class="org.springframework.security.core.userdetails.
                        UserDetailsByNameServiceWrapper">
                <constructor-arg ref="userService" />
            </bean>
        </property>
        <property name="serviceProperties" ref="serviceProperties" />
        <property name="ticketValidator">
            <bean class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
                <constructor-arg index="0" value="https://localhost:9443" />
            </bean>
        </property>
        <property name="key" value="some_id_for_this_cas_prov" />
    </bean>

</beans>
```

I will explain the elements from the previous XML a bit later. But first we need to make a couple of configuration changes in the SSL elements of our application to be able to run everything together:

1. First, from the root of the CAS runner you created (wherever the `pom.xml` from Listing 6-17 is), run the following command:

```
keytool -export -alias localhost -file CAS.crt -keystore target/jetty-ssl.keystore
```

Download from Wow! eBook <www.wowebook.com>

That command creates an X.509 certificate that you need to include in the trusted certificates of your JRE so that other applications using the same JRE automatically trust this certificate.

2. Next import the generated certificate from the previous point into the `cacerts` file in your JRE. Usually, this file resides in the `$JDK_HOME/jre/lib/security` directory. To import the certificate, run the following command from that directory:

   ```
   keytool -import -alias Local_CAS -keystore cacerts -trustcacerts -file / ... wherever you
   generated the certificate file .... /CAS.crt
   ```

3. Next, from the root of the application you will use to test, you need to run the following command so that it generates the key you need to set up its SSL environment:

   ```
   mvn clean install keytool:genkey
   ```

You don't need to import this one anywhere.

The application is now ready to be run. Restart the CAS application first. Again, from the root of the CAS-runner (wherever the `pom.xml` from Listing 6-17 is), execute `mvn jetty:deploy-war`. Then in the root of the example application execute `mvn jetty:run` and wait for the application to start. It should start without problems.

Next try to visit the URL `http://localhost:8080`. You will get the page shown in Figure 6-24 automatically, which means that the application redirected to the CAS web site to handle the authentication. Look at the URL in that figure and notice how it contains information that points back to the URL of the example application. This is the callback URL that CAS will call when authentication succeeds.

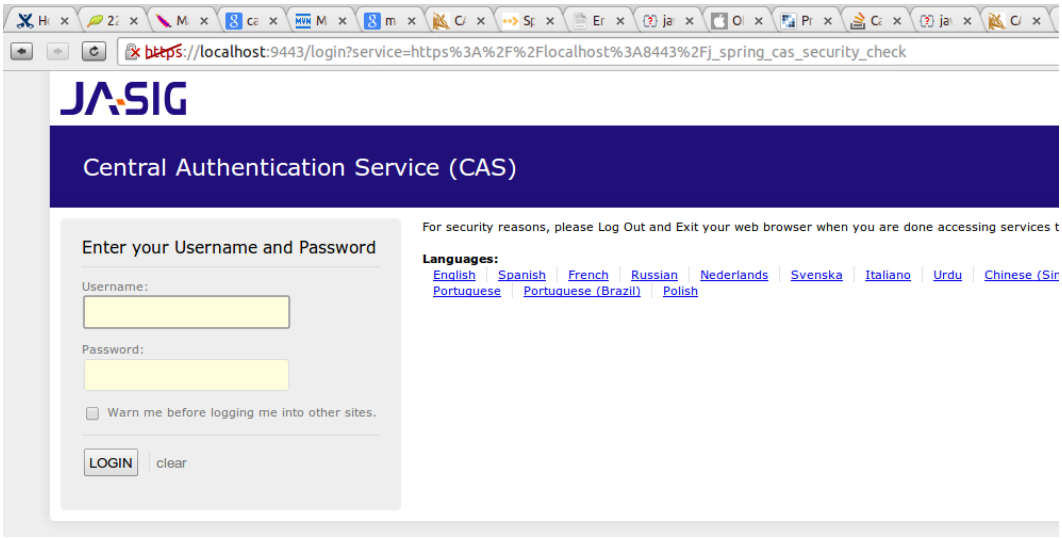


Figure 6-24. *You are redirected to CAS for secured resources*

Now log in with the username **car** and the password **car**. You are redirected back to the client application. If you access `https://localhost:8443/hello`, you are presented with the screen from Figure 6-25 showing a successful login.

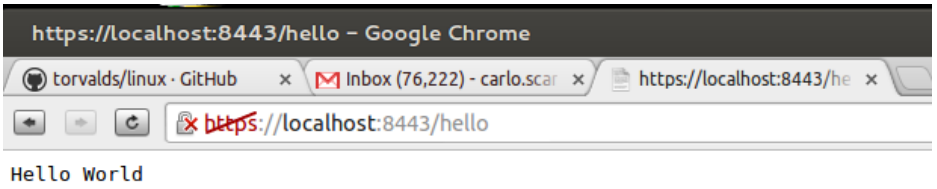


Figure 6-25. *You are redirected back to the "Hello World" page*

Now let's see exactly what is happening under the hood:

1. When you first request the URL `http://localhost:8080/hello`, the request goes through the security filter chain, as always. The `FilterSecurityInterceptor` sees that you are trying to access a secured resource, and it throws an `AccessDeniedException` exception because there is no authenticated user with the required role.

2. The `ExceptionTranslationFilter` catches the `AccessDenied` exception and calls the `CasAuthenticationEntryPoint`'s `commence` method with the request and the response. The `ExceptionTranslationFilter` calls this entry point because you configured the attribute `entry-point-ref="casEntryPoint"` in the `<http>` element in the `applicationContext-security.xml`. Of course, you also defined a "`casEntryPoint`" bean in the same configuration file. You can refer to this file back in Listing 6-21.

3. The `CasAuthenticationEntryPoint`'s `commence` method receives the request. It first constructs the URL of the service that looks something like `https://localhost:8443/j_spring_cas_security_check;jsessionid=1pivs7a82frfir23swj4wknty`. That value is built using the configured `org.springframework.security.cas.ServiceProperties` that is injected into the `CasAuthenticationEntryPoint`. You can see the definition of the `ServiceProperties` in the configuration file `applicationContext-security.xml`. As you can also see in the built URL, the method concatenates the jsesssionid, of the current user session, into the callback URL. The `CasAuthenticationEntryPoint` then creates the full CAS URL that the application needs to redirect to in order to authenticate. This URL is of the form `https://localhost:9443/login?service=https%3A%2F%2Flocalhost%3A8443%2Fj_spring_cas_security_check%3Bjsessionid%3D1pivs7a82frfir23swj4wknty`. The URL is built using the configured "`loginUrl`" property of the `CasAuthenticationEntryPoint` bean and then is concatenated with the service callback URL generated before. The application then redirects to this URL, which shows the CAS login screen.

4. After you log in to CAS, you are redirected back to the callback URL. The request arrives at the `CasAuthenticationFilter`. The first thing it does is check whether the request needs to be authenticated. In our current case, the check simply involves seeing if the requested URL is for the URL `/j_spring_cas_security_check`. It returns `true` if that is the case. In the case of this example, running it on my Linux machine the request is `https://localhost:8443/j_spring_cas_security_check?ticket=ST-1-gEMNCrUKf35LhgmdhOld-cas01.example.org`. So the filter knows it needs to process this request.

5. When attempting authentication, the `CasAuthenticationFilter` tries to extract the 'ticket' parameter from the request. You can see from the previous step that the parameter does exist. So the filter extracts this parameter and uses it as the value for the user password. The user name is the static string '`_cas_stateful_` '. A new `UsernamePasswordAuthenticationToken` is created with that username and that password. This `UsernamePasswordAuthentication` is then passed to the `ProviderManager`, which in turns passes it to the `CasAuthenticationProvider` for authentication.

6. CasAuthenticationProvider receives the Authentication object (the instance of UsernamePasswordAuthentication) and creates a new instance of CasAuthenticationToken, which is another Authentication implementation, which will replace the UsernamePasswordAuthenticationToken that arrives when the authentication process is over. But before that, the CasAuthenticationProvider uses the configured org.jasig.cas.client.validation.Cas20ServiceTicketValidator from the CasAuthenticationProvider bean to call the CAS server again to validate the received ticket. The validation URL in the CAS server to call in my current request is `https://localhost:9443/serviceValidate?ticket=ST-1-gEMNCrUKf35LhgmdhOld-cas01.example.org&service=https%3A%2F%2Flocalhost%3A8443%2Fj_spring_cas_security_check`. When this URL is called, the CAS server internally validates the ticket and the service and then it returns an XML like the one shown in Listing 6-22. That XML is parsed into an instance of org.jasig.cas.client.validation.AssertionImpl, which contains the successful status and the principal name. In the current case, it is "car."

   Listing 6-22. CAS server validation message for a ticket

```
<cas:serviceResponse xmlns:cas='http://www.yale.edu/tp/cas'>
    <cas:authenticationSuccess>
        <cas:user>car</cas:user>
    </cas:authenticationSuccess>
 </cas:serviceResponse>
```

7. The CAS authentication provider gets the org.jasig.cas.client.validation.AssertionImpl instance returned from the validator, wraps it in a CasAssertionAuthenticationToken, and calls the default configured org.springframework.security.provisioning.InMemoryUserDetailsManager's loadUserByUsername method to retrieve the configured user and its authorities from the in-memory user store (the <security:user-service> element in our configuration file).

8. `CasAuthenticationProvider` creates a new fully authenticated instance of `CasAuthenticationToken` (which is an implementation of `Authentication`) with the details returned from both the CAS server and the `UserDetailsService` implementation (`InMemoryUserDetailsManager`).

9. After you have the fully authenticated Authentication object, the flow continues as normal. The `FilterSecurityInterceptor` sees the secured resource and can match the required attributes for accessing it against the authorities in the `Authentication`. So it grants access.

10. This process can be seen graphically in Figure 6-26. Not everything is covered, just the most important parts in the interaction with the CAS server.

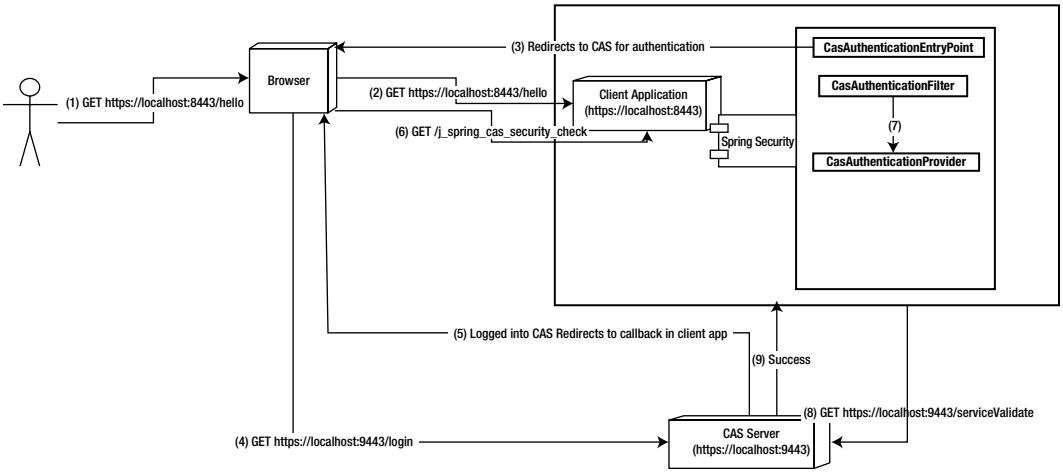


Figure 6-26. *CAS authentication process*

Integrating CAS with a Different Authentication Provider

As I said before, CAS is designed to support the integration of different back-end user storage mechanisms, including JDBC, LDAP, and others. Currently, we are using the demo version of CAS, which matches users anytime you use the same username and password as you have been doing with "car."

Because I am not going to explain CAS in any more depth than necessary, this part can be thought of as a bonus. For much more information regarding CAS, go to https://wiki.jasig.org/display/CASUM/Home.

Let's configure CAS to use file system user storage, which is probably one of the easiest things to configure. You will get a nice surprise when trying to configure CAS, and this surprise is that CAS is configured using Spring. So, many of the things you have learned regarding core Spring apply to configuring the CAS server.

1. Create a file called `users.txt` somewhere in your file system with the contents of Listing 6-23. I created it in `/tmp/`.

   Listing 6-23. users.txt with a couple of users in directory /tmp

   ```
   car::scarvarez
                 mon::scarvarez2
   ```

2. In the root of the downloaded CAS server, go to the directory `./cas-server-webapp/src/main/webapp/WEB-INF`. In the `deployerConfigContext.xml` file, replace the bean property named `"authenticationHandlers"` of the bean `authenticationManager` with the contents of Listing 6-24.

   Listing 6-24. fileAuthentication.xml. The bean for activating the file-system user-authentication mechanism

   ```
   <property name="authenticationHandlers">
       <list>
           <bean class="org.jasig.cas.adaptors.generic.FileAuthenticationHandler">
              <property name="fileName" value="file:/tmp/users.txt"/>
           </bean>
       </list>
   </property>
   ```

3. In the `pom.xml` file in the directory `./cas-server-webapp`, add the dependency from Listing 6-25 to enable the file-system support.

   Listing 6-25. Dependency on generic handlers

   ```
   <dependency>
      <groupId>org.jasig.cas</groupId>
      <artifactId>cas-server-support-generic</artifactId>
      <version>${project.version}</version>
   </dependency>
   ```

4. In the directory `./cas-server-webapp`, execute **mvn install** to re-create the war file.

5. In the `pom.xml` of the CAS runner, replace the line that points to the war file with the path to the newly generated war. The newly generated war should be in the directory relative to the CAS root `./cas-server-webapp/target/cas.war`.

6. Run both applications. The CAS runner and the example application (remember to run them with the commands `mvn jetty:deploy-war` and `mvn jetty:run`, respectively).

7. Visit the URL `https://localhost:8443/hello`. You are redirected to CAS. Log in with the username **car** and the password **scarvarez**. Then visit the URL `https://localhost:8443/hello`. You should be granted access.

That's it. You have successfully configured CAS for file-system authentication. The same (but with different authentication handlers) can be done for LDAP, JDBC, and others.

Summary

In this chapter, I showed you how you can use Spring Security's modular architecture to integrate different authentication mechanisms with relative ease. I explained some of the authentication mechanisms that come with the framework. In particular, I showed you how to authenticate your users against a database, an LDAP server, and an OpenID provider, and by using Client X.509 Certificates and leveraging Java's standard authentication system, JAAS.

This chapter focused on showing how all these different authentication providers relate to each other when they are used inside the framework. The goal was to show you that integrating new providers into the framework is simple enough for you to try. Of course, how easy it is depends on the authentication scheme that you want to plug in.

There are more authentication providers that I haven't covered in the chapter, but the main ideas tend to remain the same: create a connector into Spring Security that deals with the particulars of the integrating protocol, and adapt it to use the Spring Security model of authentication and authorization.

CHAPTER 7

Business Object Security with ACLs

This chapter will introduce access control lists (ACLs) in the context of Spring Security.

Access control lists can be thought of as an extension to the business-level security rules that we reviewed in Chapter 6. In this case, however, we'll be looking at more fine-grained rules to secure individual domain objects, instead of the relatively coarse-grained rules used to secure method calls on services.

What this means is that ACLs are in charge of securing instances of domain classes (such as a `Forum` class, a `Cart` class, and so on), while the standard method-level rules secure entry points determined by methods (like a `Service` method or a `DAO` method).

Securing domain objects with ACLs is conceptually simple. The idea is that any user will have a certain level of access (read, write, none, and so on) to each domain object. A user's level of access (*permissions*) to a particular domain object depends on the user, or the role or group to which the user belongs.

As in other chapters, I'll try to explain each concept as I walk through an example. The example I'll be working on is, as usual, a very simple and not *real-world* application pared down to focus on the concepts relevant to understanding how ACLs work.

The Security Example Application

The example application will be a simple forum system with two types of users: standard users and administrator users. Any user can create a forum entry, but only the user who created the entry can edit it; the other standard users can only read it. Administrator users can read or delete an entry, but they cannot edit it. I think this give us all the combinations we need to show the full power of ACLs. The permission logic is shown in Figure 7-1. The action is shown with a solid line and permissions are shown with a dotted line.

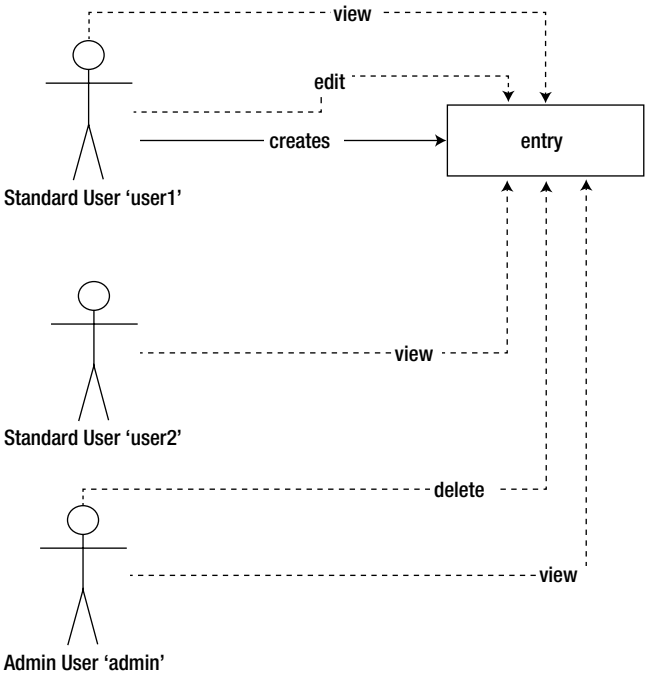


Figure 7-1. *Permission logic for forum entries*

So let's set up our example project.

The first thing we'll do is review the required database schema to support ACLs in the application. Currently, this is the only support for ACLs in Spring Security, and it needs to be configured for the whole functionality to work. Spring Security's ACL module comes with the SQL scripts necessary to define its own support in a file named `createAclSchema.sql` that currently targets the HSQL database. (There is also an alternative file called `createAclSchemaPostgres.sql` targeting the PostgreSQL database.) You can find this file in the source code of the Spring Security ACL module or inside the jar file itself: `spring-security-acl-3.1.3.RELEASE.jar`. It will be in the `src/main/resources` folder in the source code, which means it will be in the root of the `classpath`. Listing 7-1 shows the contents of this file.

Listing 7-1. The createAclSchema.sql that defines the needed tables for supporting ACL in Spring Security

```
-- ACL schema sql used in HSQLDB

-- drop table acl_entry;
-- drop table acl_object_identity;
-- drop table acl_class;
-- drop table acl_sid;

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 unique_uk_1 unique(sid,principal));
```

```
create table acl_class(
    id bigint generated by default as identity(start with 100) not null primary key,
    class varchar_ignorecase(100) not null,
    constraint unique_uk_2 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,
    entries_inheriting boolean not null,
    constraint unique_uk_3 unique(object_id_class,object_id_identity),
    constraint foreign_fk_1 foreign key(parent_object)references acl_object_identity(id),
    constraint foreign_fk_2 foreign key(object_id_class)references acl_class(id),
    constraint foreign_fk_3 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 unique_uk_4 unique(acl_object_identity,ace_order),
    constraint foreign_fk_4 foreign key(acl_object_identity) references acl_object_identity(id),
    constraint foreign_fk_5 foreign key(sid) references acl_sid(id)
);
```

To understand the meaning of these tables, you need to appreciate the main abstractions in SpringSecurity's ACL support:

- **SID (Security Identity)** An abstraction that represents a security identity in the system to be used by the ACL infrastructure. A security identity can be a user, role, group, and so forth. It maps to the table `ACL_SID`.
- **Access Control Entry (ACE)** Represents an individual permission in the particular ACL making relationships between objects, SIDs, and permissions. It maps to the table `ACL_ENTRY`.
- **Object Identity** Represents the identity of an individual domain object instance. These are the entities on which the permissions are set. It maps to the table `ACL_OBJECT_IDENTITY`.

Let's break down the database tables into their main attributes to see their meaning. (I won't explain the ID attributes because they are the surrogate identifiers of each row in the pertinent table, and they serve to, well, identify the particular entry.) Figure 7-2 shows the tables in graphical form followed by the explanation of the attributes.

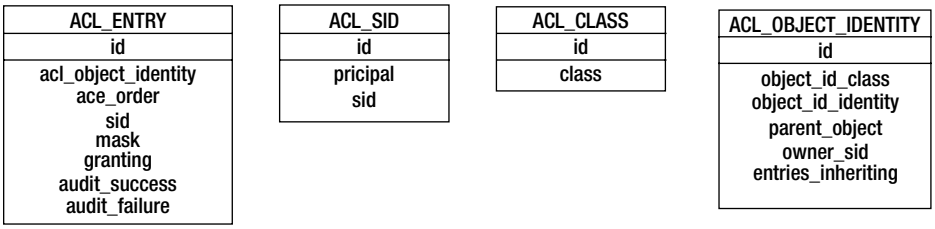


Figure 7-2. *Tables in the ACL schema*

Here are the main attributes (columns) in the tables from the previous figure:

- Table `ACL_SID`
  - `principal`: A Boolean that indicates if the particular SID is a principal or not. Remember that the SID can represent a group or role.
  - `sid`: The name of the SID. It can be a username of a principal, a role name, and so on.
- Table `ACL_CLASS`
  - `class`: The name of the class of the domain objects you can have in the ACL system.
- Table `ACL_OBJECT_IDENTITY`
  - `object_id_identity`: A number that identifies uniquely (when combined with the `object_id_class`) the particular instance of the domain object you are representing.
  - `object_id_class`: A foreign key to the table `ACL_CLASS` that identifies the class of the domain object you are representing.
  - `parent_object`: Allows for a hierarchy of domain objects by creating a recursive relation with itself. So permission can be shared between the objects in the hierarchy. It is a foreign key to itself.
  - `owner_sid`: The SID owning this particular domain object.
  - `entries_inheriting`: A Boolean that specifies if, in a hierarchy of domain objects, permissions are inherited between the objects.
- Table `ACL_ENTRY`
  - `acl_object_identity`: The identity of the object. It's a reference to a row in table `ACL_OBJECT_IDENTITY`.
  - `sid`: Identifies the SID in this ACE. It's a reference to a row in table `ACL_SID`.
  - `mask`: The actual permission. Permissions in ACL support are given as a bitmask represented with an integer value. In code, the default permissions are defined in the class `org.springframework.security.acls.domain.BasePermission`, which is reproduced in Listing 7-2. You can easily create your own class and define extra permissions if you want to do so.

Listing 7-2. The BasePermission class with the default permissions to use in the ACL system

```
package org.springframework.security.acls.domain;

import org.springframework.security.acls.model.Permission;
```

```
public class BasePermission extends AbstractPermission {
    public static final Permission READ = new BasePermission(1 << 0, 'R'); // 1
    public static final Permission WRITE = new BasePermission(1 << 1, 'W'); // 2
    public static final Permission CREATE = new BasePermission(1 << 2, 'C'); // 4
    public static final Permission DELETE = new BasePermission(1 << 3, 'D'); // 8
    public static final Permission ADMINISTRATION =
new BasePermission(1 << 4, 'A'); // 16

    protected BasePermission(int mask) {
       super(mask);
    }

    protected BasePermission(int mask, char code) {
        super(mask, code);
    }
}
```

- `granting`: A Boolean that determines if the particular ACE is for granting or, if false, for denying access according to the rules defined in the entry.

You can see that there are not a lot of tables defined in the schema, but it is important that you understand them individually and their relationships. Figure 7-3 shows the ER (Entity-Relationship) diagram for this data model to help you understand them better. The diagram describes the meanings of the relationships between tables in simple terms.

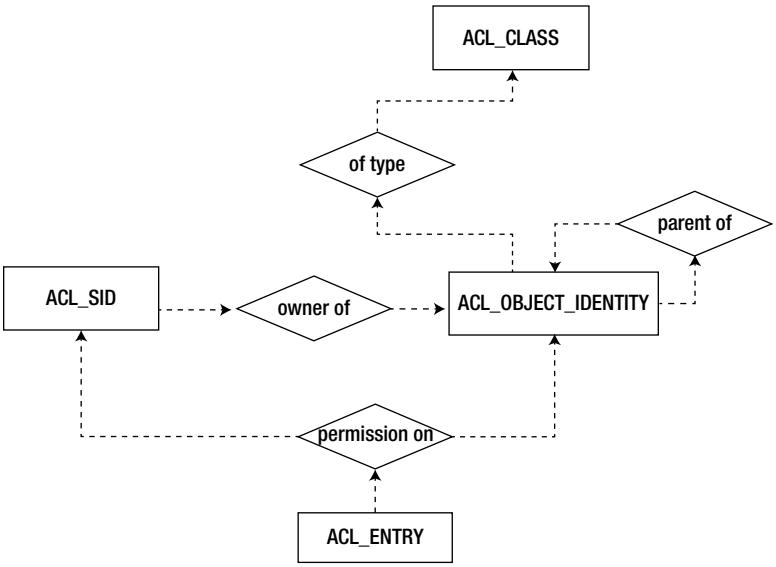


Figure 7-3. *The ER diagram for the ACL security model*

The preceding code shows the existing permissions with bitmasks represented as the integers 1, 2, 4, 8, and 16 for Read, Write, Create, Delete, and Administration, respectively. There are 32 bits available for permissions, and the default permissions set bit 0, bit 1, bit 2, bit 3, and bit 4, respectively, to each integer value mentioned before. This class can be replaced with a custom `Permission` implementation class that includes different permissions, or it can be extended to include more permissions.

Download from Wow! eBook <www.wowebook.com>

In theory, working this way allows for the permissions to be easily combined to create composed permissions by adding two of them together. For example, you could combine READ and WRITE permissions by summing their respective values (1+2 = 3) to grant both READ and WRITE permissions. However, in practice, it doesn't really work like this. The class `org.springframework.security.acls.domain.DefaultPermissionGrantingStrategy`, which is the one in charge of evaluating the permissions required for an object against the permissions stored in the list of ACEs (Access Control Entries), will do comparisons for exact matches. So you normally have to include an ACE for READ and another one for WRITE.

You should now have an understanding of the main concepts used in ACLs and how they map to the database schema that the framework itself provides. Next, let's set up the application itself. Again, we'll be using Maven for this. From some directory in your shell, execute the command `mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes -DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.0 -DgroupId=com.apress.pss -DartifactId=acl-example -Dversion=1.0-SNAPSHOT`. Then, in the generated `pom.xml` file, add all the required dependencies that I show you in Listing 7-3, which includes the new ACL dependency.

Listing 7-3. Maven dependencies in the Spring Security ACL project

```
<dependency>
       <groupId>javax.servlet</groupId>
       <artifactId>javax.servlet-api</artifactId>
       <version>3.0.1</version>
   </dependency>

   <dependency>
       <groupId>junit</groupId>
       <artifactId>junit</artifactId>
       <version>4.5</version>
       <scope>test</scope>
   </dependency>
   <dependency>
       <groupId>org.springframework.security</groupId>
       <artifactId>spring-security-core</artifactId>
       <version>3.1.2.RELEASE</version>
   </dependency>
   <dependency>
       <groupId>org.springframework.security</groupId>
       <artifactId>spring-security-config</artifactId>
       <version>3.1.2.RELEASE</version>
   </dependency>
   <dependency>
       <groupId>org.springframework.security</groupId>
       <artifactId>spring-security-web</artifactId>
       <version>3.1.2.RELEASE</version>
   </dependency>
   <dependency>
       <groupId>org.springframework.security</groupId>
       <artifactId>spring-security-taglibs</artifactId>
       <version>3.1.2.RELEASE</version>
   </dependency>
   <dependency>
       <groupId>org.springframework.security</groupId>
       <artifactId>spring-security-acl</artifactId>
       <version>3.1.2.RELEASE</version>
   </dependency>
```

```
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-webmvc</artifactId>
        <version>3.0.6.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.aspectj</groupId>
        <artifactId>aspectjweaver</artifactId>
        <version>1.7.0</version>
    </dependency>

    <dependency>
        <groupId>commons-logging</groupId>
        <artifactId>commons-logging</artifactId>
        <version>1.1.1</version>
    </dependency>
    <dependency>
        <groupId>commons-codec</groupId>
        <artifactId>commons-codec</artifactId>
        <version>1.3</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet.jsp</groupId>
        <artifactId>jsp-api</artifactId>
        <version>2.1</version>
    </dependency>
    <dependency>
        <groupId>javax.servlet</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
    <dependency>
        <groupId>javax.annotation</groupId>
        <artifactId>jsr250-api</artifactId>
        <version>1.0</version>
    </dependency>
    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.2.8</version>
    </dependency>
    <dependency>
        <groupId>net.sf.ehcache</groupId>
        <artifactId>ehcache-core</artifactId>
        <version>2.6.0</version>
</dependency>
```

You also need to include in the `pom.xml` file, the configuration for the jetty plugin we have been using up until now. So, in the plugins section of your `pom.xml` file, add the code from Listing 7-4.

Listing 7-4. Maven jetty plugin in pom.xml

```
<plugin>
 <groupId>org.mortbay.jetty</groupId>
 <artifactId>jetty-maven-plugin</artifactId>
 <version>8.1.1.v20120215</version>
 <configuration>
     <connectors>
         <connector implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
             <port>8080</port>
             <maxIdleTime>60000</maxIdleTime>
         </connector>
     </connectors>
 </configuration>
</plugin>
```

Next, let's create our first configuration file, the `web.xml`. Copy the one from Listing 7-5.

Listing 7-5. web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
    <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
           /WEB-INF/applicationContext-acl.xml
           /WEB-INF/applicationContext-security.xml
        </param-value>
    </context-param>

    <servlet>
        <servlet-name>acl-example</servlet-name>
        <servlet-class> org.springframework.web.servlet.DispatcherServlet </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>acl-example</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
```

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

The next file to create is the ACL configuration file, so let's do that. This will be a very standard default ACL file. Later, I'll show you that some things could be changed to adapt to different needs. In the `WEB-INF` folder, create a file named `applicationContext-acl.xml` with the contents from Listing 7-6. After the listing, you will read a description of the relevant parts.

Listing 7-6. The applicationContext-acl.xml with the ACL configuration needed

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xmlns:util=http://www.springframework.org/schema/util
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
    http://www.springframework.org/schema/util
    http://www.springframework.org/schema/util/spring-util-3.0.xsd">

<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
   <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
   <property name="url" value="jdbc:hsqldb:mem:aclexample"/>
   <property name="username" value="sa"/>
   <property name="password" value=""/>
</bean>

<bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
   <property name="dataSource" ref="dataSource"/>
</bean>

<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
   <property name="dataSource" ref="dataSource"/>
</bean>
```

```
<tx:annotation-driven transaction-manager="transactionManager" />
<bean id="aclCache" class="org.springframework.security.acls.domain.EhCacheBasedAclCache">
  <constructor-arg>
     <bean class="org.springframework.cache.ehcache.EhCacheFactoryBean">
       <property name="cacheManager">
          <bean class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean"/>
       </property>
       <property name="cacheName" value="aclCache"/>
     </bean>
  </constructor-arg>
</bean>

<bean id="lookupStrategy" class="org.springframework.security.acls.jdbc.BasicLookupStrategy">
   <constructor-arg ref="dataSource"/>
   <constructor-arg ref="aclCache"/>
   <constructor-arg ref="aclAuthorizationStrategy"/>
   <constructor-arg>
      <bean class="org.springframework.security.acls.domain.ConsoleAuditLogger"/>
   </constructor-arg>
</bean>

<bean id="aclAuthorizationStrategy"
     class="org.springframework.security.acls.domain.AclAuthorizationStrategyImpl">
  <constructor-arg>
   <list>
    <bean class="org.springframework.security.core.authority.SimpleGrantedAuthority">
      <constructor-arg value="ROLE_ADMIN"/>
    </bean>
    <bean class="org.springframework.security.core.authority.SimpleGrantedAuthority">
      <constructor-arg value="ROLE_ADMIN "/>
    </bean>
    <bean class="org.springframework.security.core.authority.SimpleGrantedAuthority">
      <constructor-arg value="ROLE_ADMIN "/>
    </bean>
   </list>
 </constructor-arg>
</bean>

<bean id="aclService" class="org.springframework.security.acls.jdbc.JdbcMutableAclService">
   <constructor-arg ref="dataSource"/>
   <constructor-arg ref="lookupStrategy"/>
   <constructor-arg ref="aclCache"/>
</bean>
</beans>
```

Let's give an overview of each of the beans defined in Listing 7-7. Later, when using the application, I'll offer a more in-depth explanation of the inner works of the framework and exactly what is going on both at startup and at execution time.

Listing 7-7. The bean that creates the database tables needed for working with ACLs

```
<bean id=”databaseSeeder” class=”com.apress.pss.acl.DatabaseSeeder” >
        <constructor-arg ref=”jdbcTemplate” />
</bean>
```

The first three beans are not ACL specific because they simply define a data source, a JDBC template, and a transaction manager. These beans will be used by the ACL infrastructure to access the ACL-specific tables and data in the schema.

■ **Note** The support beans for working with databases that we defined use classes from the Spring Framework core libraries. If you have worked with Spring before, you most likely have encountered and used these classes. You can find information about Spring database support on the official site http://static.springsource.org/spring/docs/3.0.x/reference/jdbc.html.

The next bean I defined is the one with the ID "aclCache". You need to have a cache implementation for the ACL system to work. Its function is to make ACL accesses faster by caching the ACLs and not accessing the database for every single query. The implementation we are using in the bean definition (EhCacheBasedAclCache) is the only one defined currently in the ACL module, and it is a very simple implementation that delegates to EhCache (http://ehcache.org/).

The next bean is the one with the ID "lookupStrategy". A lookup strategy is in charge of retrieving the object identities (as explained before) and the ACLs that apply to each of those object identities. The implementation we are using (BasicLookupStrategy) is again the only one defined in the framework. It will try to load the ACLs from the cache, and if they are not there, it will look them up on the database and cache the results. The lookup to the database is done in batches so that many items can be loaded to the cache at the same time to improve performance.

The next bean, with the ID "aclAuthorizationStrategy", defines an org.springframework.security.acls.domain.AclAuthorizationStrategy. The goal of an AclAuthorizationStrategy is to determine if a particular principal is able to execute administrative activities in the ACL infrastructure itself. For example, the currently defined permissions are "CHANGE_OWNERSHIP", "CHANGE_AUDITING", and "CHANGE_GENERAL". The default implementation we are using in the bean definition file receives in the constructor three instances of GrantedAuthority, which determine the entities that will have the permissions mentioned before. In our example—and, in general, this is the most common approach—I am setting the ROLE_ADMIN to be the one that will have the mentioned permissions. In the default implementation, note that the owner of a particular ACL is also allowed the permissions "CHANGE_OWNERSHIP" and "CHANGE_GENERAL" in the particular ACL.

The last bean, with the ID "aclService", is the main component of the whole framework. The interfaces AclService and MutableAclService allow access to all the ACL-related operations, such as reading ACLs by ID, creating ACLs, deleting ACLs, and updating ACLs.

These are all the beans needed in the application to work with ACLs. Let’s continue configuring the application. As always, you need to make the file from Listing 7-6 load up with the application. You already know how to do this by modifying the "contextConfigLocation" context-param in the web.xml. In case it is needed, the web.xml I provided already has the appropriate configuration.

Now let’s make a bean to create the database schema for ACL. You normally will not do this in a standard application, but we’ll do it here just for convenience. We define the bean from Listing 7-7 in the file applicationContext-acl.xml and create the class from Listing 7-8 in the corresponding package. This class (which is named DatabaseSeeder) uses a copy of the provided "createAclSchema.sql" file with uncommented drop tables to create the required tables in the database. (The file is provided in the source code of the book. It is named customCreateAclSchema.sql and is located in the src/main/resources folder.) If you want to create it yourself, simply take the one that comes with the framework and uncomment the beginning lines that have the commented “drop table” statements.) (Remember that the database is an in-memory HSQL database named aclexample that we

defined in the dataSource bean.) This bean will be instantiated by the context as normal when starting up, and the constructor will take care of setting up the schema.

Listing 7-8. The DatabaseSeeder class that executes the schema creation

```
package com.apress.pss.acl;

import java.io.IOException;

import org.springframework.core.io.ClassPathResource;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.util.FileCopyUtils;

public class DatabaseSeeder {
    public DatabaseSeeder(JdbcTemplate jdbcTemplate) throws IOException{
        String sql = new String(FileCopyUtils.copyToByteArray(
            new ClassPathResource("customCreateAclSchema.sql").
            getInputStream()));
        jdbcTemplate.execute(sql);
    }
}
```

Now we can create ACLs for domain objects, so let's create our classes to set up the whole process. This will involve a bit of coding because we'll need a controller, service, domain object and JSP file. You can see these files in Listings 7-9, 7-10, 7-11, and 7-12, respectively.

Listing 7-9. The ForumController entry point into post creation

```
package com.apress.pss.acl.controllers;

import java.util.HashMap;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

import com.apress.pss.acl.domain.Post;
import com.apress.pss.acl.services.ForumService;

@Controller
@RequestMapping("/forum")
public class ForumController {
    @Autowired
    private ForumService forumService;
    @RequestMapping(method = RequestMethod.POST, value = "/post")
```

```
    public ModelAndView createPost(@RequestBody String postContent){
        forumService.createPost(new Post(postContent));
        return showForm();
    }

    @RequestMapping(method = RequestMethod.GET, value = "/")
    public ModelAndView showForm(){
        Map<String, Object> model = new HashMap<String, Object>();
        model.put("posts", forumService.getPosts());
        return new ModelAndView("form",model);
    }
}
```

Listing 7-10. ForumServiceImp,. which currently just allows for creating posts in an in-memory map and creating the ACL in the database

```
package com.apress.pss.acl.services;

import java.util.HashMap;
import java.util.Map;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.acls.domain.BasePermission;
import org.springframework.security.acls.domain.GrantedAuthoritySid;
import org.springframework.security.acls.domain.ObjectIdentityImpl;
import org.springframework.security.acls.domain.PrincipalSid;
import org.springframework.security.acls.model.MutableAcl;
import org.springframework.security.acls.model.MutableAclService;
import org.springframework.security.acls.model.ObjectIdentity;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import org.springframework.security.core.userdetails.User;

import com.apress.pss.acl.domain.Post;

@Service
public class ForumServiceImpl implements ForumService {
    @Autowired
    private MutableAclService mutableAclService;
    private Map<Integer, Post> postStore = new HashMap<Integer, Post>();

    @Transactional
    @Secured({"ROLE_USER","ROLE_ADMIN"})
    public void createPost(Post post) {
        Integer id = new Integer(Math.abs(post.hashCode()));
        ObjectIdentity oid = new ObjectIdentityImpl(Post.class, id);
        MutableAcl acl = mutableAclService.createAcl(oid);

        User user = (User)SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        acl.insertAce(0, BasePermission.ADMINISTRATION,
```

```
                new PrincipalSid(
                user.getUsername()), true);

        acl.insertAce(1, BasePermission.DELETE,
                new GrantedAuthoritySid(
                "ROLE_ADMIN"), true);
        acl.insertAce(2, BasePermission.READ,
                new GrantedAuthoritySid(
                "ROLE_USER"), true);
        mutableAclService.updateAcl(acl);
        post.setId(id);
        postStore.put(id, post);
    }

    public Map<Integer, Post> getPosts(){
        return postStore;
    }

    public void setMutableAclService(MutableAclService mutableAclService) {
        this.mutableAclService = mutableAclService;
    }

}
```

Listing 7-11. The Post class. A simple domain model that we'll use for trying ACL rules

```
package com.apress.pss.acl.domain;

public class Post {

    private String content;
    private Integer id;

    public Post(String postContent) {
        this.content = postContent;
    }

    public String getContent() {
        return content;
    }

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
```

```
        int result = 1;
        result = prime * result + ((content == null) ? 0 : content.hashCode());
        result = prime * result + ((id == null) ? 0 : id.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Post other = (Post) obj;
        if (content == null) {
            if (other.content != null)
                return false;
        } else if (!content.equals(other.content))
            return false;
        if (id == null) {
            if (other.id != null)
                return false;
        } else if (!id.equals(other.id))
            return false;
        return true;
    }

}
```

Listing 7-12. The form.jsp file with a form for a new post and a list of existing posts

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Posts</title>
</head>
<body>
<form method="post" action="/forum/post">
        New Post Content: <input type="text" name="postContent"/><br/>
        <input type="submit"/>
    </form>
    <c:forEach items="${posts.keySet()}" var="key">
            ${posts[key].content} <br />
    </c:forEach>
</body>
</html>
```

Download from Wow! eBook <www.wowebook.com>

The first version of the controller shown in Listing 7-9 allows us to show the form to create new posts and has the action to actually post these new posts for them to be saved. Later, we'll add to this controller to support more functionality.

Listing 7-10 has the core functionality for creating an ACL for a domain object instance. In this case, we are creating a new `Post` object. The `Post` instance is stored in a memory map (which is not a realistic example, but is enough to show the functionality). The ACL is created using the hash code of the `Post` instance as the ID of the domain object. Then we create three ACEs (access control entries) for the ACL. In the first one, we specify that the `ADMINISTRATION` permission is granted to the creator of the `Post`, given by the `SecurityContext`'s principal. The second one gives `DELETE` permission to any user with the role `ROLE_ADMIN`, and the third one gives the `READ` permission to any authenticated user with the role `ROLE_USER`. You can also see that the method is marked as `@Transactional`. This is a requirement of the `JdbcMutableAclService` so that it can execute all the SQLs in the context of a transaction. The `ForumService` interface that is implemented by `ForumServiceImpl` is simply defined as the following:

```
package com.apress.pss.acl.services;

import java.util.Map;

public interface ForumService {
    void createPost(Post post);
    Map<Integer, Post> getPosts();
}
```

■ **Note** Keep in mind that the last code sample of the class `ForumServiceImpl` is just an example of the functionality for populating ACLs for a particular domain object. I'm saying this because in a real application, you would probably move the security-related code away from the core business methods and not mix them together the way we are doing here. You could create an aspect to deal with this or simply another helper service you can call to take care of all the ACL functionality.

After you put the `form.jsp` file in the `WEB-INF/views` directory, the application is almost ready to run. We just need the files `acl-example-servlet.xml` and `applicationContext-security.xml` from Listings 7-13 and 7-14, which should go in the `WEB-INF` directory.

Listing 7-13. The acl-example-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-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-3.1.xsd">
```

```
    <context:component-scan base-package="com.apress.pss.acl.controllers" />
    <mvc:annotation-driven />

    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name = "prefix" value="/WEB-INF/views/" />
        <property name = "suffix" value=".jsp" />
    </bean>

</beans>
```

Listing 7-14. The applicationContext-security.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">
    <security:http auto-config="true" />
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user authorities="ROLE_USER" name="car"
                    password="scarvarez" />
                <security:user authorities="ROLE_ADMIN" name="mon"
                    password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
    <context:component-scan base-package="com.apress.pss.acl.services" />
    <security:global-method-security
        secured-annotations="enabled" />

</beans>
```

When you start the application, navigate to the URL `http://localhost:8080/forum/` and log in with username **car**, and the password **scarvarez**. You will be shown a very small form with just a text box. You can create a new form there. When you submit the form, it will be stored on the map and the corresponding ACL will be stored on the database. (You won't see that on the screen, but it is happening. You could use a SQL client to look at it.) What is happening under the covers inside the framework is simply the execution of some SQL scripts that are in charge of populating the tables mentioned before with the corresponding data. The ACL system offers the APIs that can be used to manipulate this data using classes, which you can see in the `ForumServiceImpl` class. Internally, all the requests are translated to SQL instructions against the configured database. It's also worth mentioning that among the internal workings of the framework is the use of the authentication's principal as the owner of the ACL and the caching of the ACLs after they have been persisted to the database. `JdbcMutableAclService` internally uses Spring's own `JdbcTemplate` and batch updates with prepared statements.

Accessing Secured Objects

The next logical step is to make sure the rules are actually working by trying to access the created posts. To do that, we'll set up a couple more users and finish our service and controller so that they handle the new options. Remember that we are re-creating the database every time we restart the app. (Again, that's not what we want in a production environment.) Also, the posts are in an `in-memory map`, so they are lost when we shut down the application. So let's make all the code changes now step by step.

The first thing we'll do is add an `org.springframework.security.acls.AclEntryVoter` to the configuration of our application. `AclEntryVoter` is an implementation of `AccessDecisionVoter` like the ones you studied before (`RoleVoter`, and so forth), which votes whether to grant or deny access based on the rules given by the ACL configuration of domain objects. You need to create an `AclEntryVoter instance` for each of the operations you want ACLs to vote on. For example, in our case, we'll create three voters: one for voting on reading access, one for voting on update permission, and one for voting on delete permission. Add these voters in the file `applicationContext-acl.xml`. Listing 7-15 shows the definition of these three voters, followed by a more comprehensive explanation of their work.

Listing 7-15. The AclEntryVoter(s) that correspond to the delete, read, and update actions

```
<bean id="aclDeletePostVoter" class="org.springframework.security.acls.AclEntryVoter">
        <constructor-arg ref="aclService" />
        <constructor-arg value="ACL_POST_DELETE" />
        <constructor-arg>
            <list>
                <util:constant
                    static-field="org.springframework.security.acls.domain.BasePermission.DELETE" />
            </list>
        </constructor-arg>
        <property name="processDomainObjectClass"
            value="com.apress.pss.acl.domain.Post" />
    </bean>

    <bean id="aclUpdatePostVoter" class="org.springframework.security.acls.AclEntryVoter">
        <constructor-arg ref="aclService" />
        <constructor-arg value="ACL_POST_UPDATE" />
        <constructor-arg>
            <list>
                <util:constant
                    static-field="org.springframework.security.acls.domain.BasePermission.
                    ADMINISTRATION" />
            </list>
        </constructor-arg>
        <property name="processDomainObjectClass"
            value="com.apress.pss.acl.domain.Post" />
    </bean>

    <bean id="aclReadPostVoter" class="org.springframework.security.acls.AclEntryVoter">
        <constructor-arg ref="aclService" />
        <constructor-arg value="ACL_POST_READ" />
        <constructor-arg>
            <list>
                <util:constant
                    static-field="org.springframework.security.acls.domain.BasePermission.READ" />
            </list>
```

```
        </constructor-arg>
        <property name="processDomainObjectClass"
            value="com.apress.pss.acl.domain.Post" />
    </bean>
```

We have three voter beans; however, to analyze one is to analyze them all. When you define an `AclEntryVoter`, you need to pass three arguments to its constructor. The first argument is a reference to the `AclService` we defined earlier, and the second parameter will be mapped to a config attribute name. I talked about config attributes earlier in the book, so here I'll just say that they are the attributes that Spring Security looks for in the `@Secured` annotation in order to use them when intercepting methods. The `AccessDecisionManager` has access to this value and as well as to the different voters that decide if they support a particular config attribute. For example, if you have an annotation like `@Secured("ROLE_USER")` and you have `RoleVoter` configured, the voter will do its work because the `RoleVoter` supports, by default, any config attribute that starts with `ROLE_`. In the case of the `AclEntryVoter`, we are specifying exactly which config attribute that particular voter will support. In our example, the first bean will support annotations like `@Secured("ACL_POST_DELETE")`, the second bean will support annotations like `@Secured("ACL_POST_UPDATE")`, and the third will support annotations like `@Secured("ACL_POST_READ")`.

The third parameter that we pass to the `AclEntryVoter` constructor is the permission needed to allow access to the particular operation we are trying to perform on the object. For example, the scenario for the first `AclEntryVoter` bean is as follows: We have a method annotated with `@Secured("ACL_POST_DELETE")` that receives a `Post` instance as a parameter. When the method is called, the `AclEntryVoter` receives the `Post` object and then retrieves the principal from the authentication and evaluates its permissions against the `Post`'s ACL to see if it has the required permission—in this case, `BasePermission.ADMINISTRATION`. If it does, it will vote to grant access; if it doesn't, it will vote to deny access. I'll explain this in more depth a little later when executing the application.

For these voters to work, you need to add them to the `AccessDecisionManager`. Currently, you are using the default `AccessDecisionManager`, so you need to define an explicit one in the application context with the voters injected so that you can use those voters in the application. You do this in the `applicationContext-security.xml` file. It's as simple as adding the two beans shown in Listing 7-16.

Listing 7-16. The beans for using a custom AccessDecisionManager

```
<security:global-method-security
        secured-annotations="enabled" access-decision-manager-ref="customAccessDecisionManager" />

    <bean id="customAccessDecisionManager" class="org.springframework.security.access.vote.
    AffirmativeBased">
        <property name="decisionVoters">
            <list>
                <ref bean="aclDeletePostVoter"/>
                <ref bean="aclUpdatePostVoter"/>
                <ref bean="aclReadPostVoter"/>
            </list>
        </property>
    </bean>
```

There's nothing special in this listing. We are using an affirmative-based access decision manager, and we are injecting the three voters in the `decisionVoters` property.

Next, let's create our delete action in the service. The `deletePost` method in `ForumServiceImpl` looks like Listing 7-17. Remember to update the `ForumService` interface as well with the new method.

Listing 7-17. The deletePost method in ForumServiceImpl

```
@Transactional
    @Secured("ACL_POST_DELETE")
    public void deletePost(Post post){
        ObjectIdentity oid = new ObjectIdentityImpl(Post.class, post.getId());
        mutableAclService.deleteAcl(oid, true);
        postStore.remove(postStore.get(post.getId()));
    }
```

It's a very simple method. Worth noting is the `@Secured` annotation. If you recall our previous definition of the voters, you should be able to see that the invocation of this method will be supported by the first voter bean we defined (`aclDeletePostVoter`). Let's try this out and see what happens. Before doing so, you need to update the `ForumController`, adding the method from Listing 7-18. You also need to replace the `form.jsp` file with the content of Listing 7-19.

Listing 7-18. The deletePost in ForumController

```
@RequestMapping(method = RequestMethod.POST, value = "/post/delete")
    public ModelAndView deletePost(@RequestParam Integer postId){
        Post post = new Post("non-relevant");
        post.setId(postId);
        forumService.deletePost(post);
        return showForm();
    }
```

Listing 7-19. The form.jsp updated to allow the deleting of posts

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Posts</title>
</head>
<body>
<form method="post" action="/forum/post">
        New Post Content: <input type="text" name="postContent"/><br/>
        <input type="submit"/>
    </form>
    <c:forEach items="${posts.keySet()}" var="key">
    <form method="post" action="/forum/post/delete">
            ${posts[key].content} <br />
            <input type="hidden" value="${key}" name="postId"/>
            <input type="submit" value="delete"/>
        </form>
    </c:forEach>
</body>
</html>
```

If I restart the application now, visit the URL `http://localhost:8080/forum/`, and create a post, it will look like Figure 7-4.



Figure 7-4. *Screen showing a created post*

Then if I click the delete button, the application redirects me to the familiar login screen shown in Figure 7-5.



Figure 7-5. *The default Spring Security login page*

This is what happened here: When I clicked the delete button, the request got all the way to the `AffirmativeBased` access decision manager, as I explained in previous chapters. The access decision manager iterates through its configured `AccessDecisionVoter(s)`—in this case, the `AclEntryVoter(s)` we injected explicitly. One of the voters will get into action—the one dealing with `ACL_POST_DELETE` config attribute. The voter traverses the `MethodInvocation` object of the method that was just intercepted, and it iterates through its parameters looking for parameters that are of the type configured in the `processDomainObjectClass` property of the `AclEntryVoter`, which in our case is `com.apress.pss.acl.domain.Post`. If an object of this type isn't found on the parameter list of the method, an `AuthorizationServiceException` is thrown informing you of this. In our case, it is found, and it is actually the only parameter that the `deletePost` method expects. If the object that arrives in this parameter is null, the voter will simply abstain from voting.

The next thing the voter does is try to retrieve an `ObjectIdentity` instance from the domain object. It does this by using an `ObjectIdentityRetrievalStrategy` that is configured by default and whose only implementation is `org.springframework.security.acls.domain.ObjectIdentityRetrievalStrategyImpl`. This strategy simply invokes the constructor of `ObjectIdentityImpl` that receives a domain object as its only parameter. This constructor in `ObjectIdentityImpl` assumes that the domain object provides a `getId` method to be able to retrieve the identifier for it. If the method is not there, a corresponding exception is thrown. Our `Post` class has such a method, so this works fine—invoking that method and setting the return value as the identifier of the `ObjectIdentity`.

After obtaining the `ObjectIdentity`, the voter tries to retrieve the SIDs from the `Authentication` object. For this, the voter uses a `SidRetrievalStrategy`, whose sole implementation (`SidRetrievalStrategyImpl`) retrieves both the authorities (for example, `ROLE_USER`) of the `Authentication` object plus the principal. So it will have an instance of `PrincipalSid` and as many instances of `GrantedAuthoritySid` as the authenticated user has authorities.

The next step for the voter is to retrieve the actual ACL for that particular `ObjectIdentity` and the SIDs. It does this with the help of the configured `AclService`, which I already talked about.

In the next step, the retrieved ACL is consulted by its `isGranted` method to see whether or not access should be granted. This method receives the required permission (as defined by the third constructor argument of the beans of type `AclEntryVoter`, which in the case of `DELETE` had the value `org.springframework.security.acls.domain.BasePermission.DELETE`) and the list of SIDs obtained before. The ACL, in turn, delegates to an instance of `PermissionGrantingStrategy` (actually, to the implementation `DefaultPermissionGrantingStrategy`) to make the final call on whether the SIDs have the required permissions on the domain object. `PermissionGrantingStrategy`'s sole implementation, `DefaultPermissionGrantingStrategy`, simply iterates through the list of permissions, the list of SIDs, and the list of ACEs in the ACL. It compares the permissions one by one against the permissions on the ACE,

and it compares the SIDs against the SIDs on the ACE. If it finds a match, it allows access; if it doesn't find any match, it rejects access by throwing an exception, which is caught by the voter to return `ACCESS_DENIED`. In our case, because the current authenticated user is `ANONYMOUS`, this whole flow is what happens.

On the login page, I log in now with the username **car** and the password **scarvarez** (which is one of the users I defined in the `applicationContext-security.xml` file) and try to delete the post. I get an Access Denied page. The same thing as before is happening: the logged-in user *car* and its roles `ROLE_USER` don't match the rules required to execute the `deletePost` action on that particular post. As I said before, only a user with role `ROLE_ADMIN` can delete the post. Next, I try that out.

First I log out by visiting the URL `http://localhost:8080/j_spring_security_logout`, and then I log in with the username **mon** and the password **scarvarez** and try to delete the post again. This time, the `Acl.isGranted` method returns true and the voter returns `ACCESS_GRANTED` because the match exists between the role of the authenticated user and the authorities required to perform the required action. This means that the code execution will finally reach the `deletePost` method in the `ForumServiceImpl`. This is a very simple method that removes the entry from the ACL for that object and, of course, deletes the object from the store. (Remember, we are using a `java.util.Map` as our in-memory store, which is not the most realistic simulation but works for the purposes of the example.) Figure 7-6 shows the most important aspects of the process I just explained.

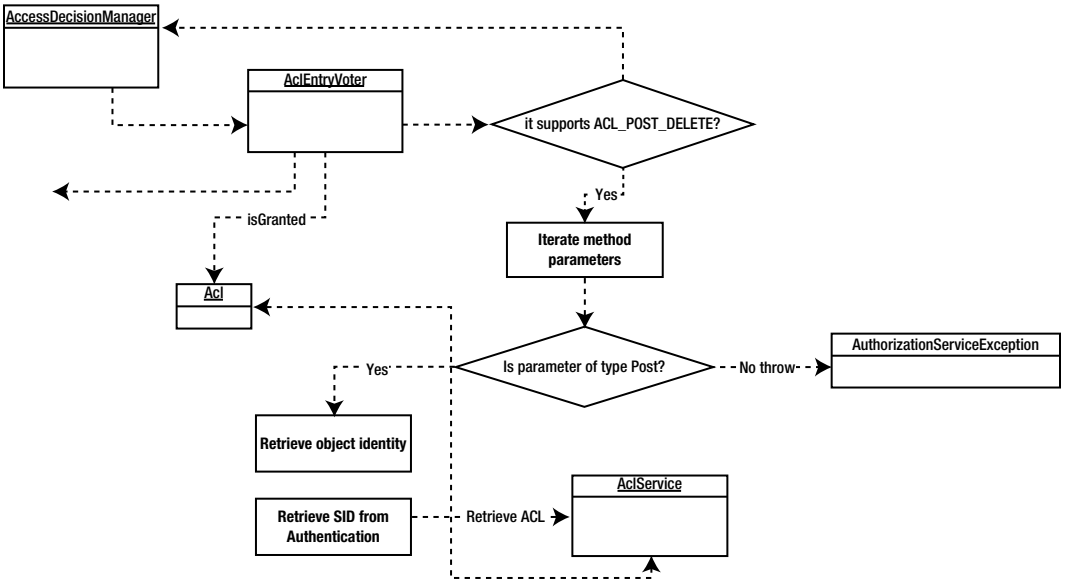


Figure 7-6. *A high-level overview of AclEntryVoter*

Filtering Returned Objects

So we successfully tested the `DELETE` action with a `Post` domain instance that should be allowed only to `ROLE_ADMIN` users. The other two actions follow a similar pattern. Let's secure the `READ` one so that only users with role `ROLE_USER` can read them. (Users with `ROLE_ADMIN` need to have `ROLE_USER` as well to read the posts. Normally, your users wouldn't need to have both roles defined, but for the sake of the example, it is OK to do it this way.) However, if the post was created by a user with role `ROLE_ADMIN`, it will be readable only by users with `ROLE_ADMIN`. First of all, let's create another user with both roles: `ROLE_USER` and `ROLE_ADMIN`. In the file `applicationContext-security.xml` in the `<user-service>`, we add this user: `<security:user authorities="ROLE_USER,ROLE_ADMIN" name="bea" password="scarvarez" />`.

In our example, we want to secure the method `getPosts` in the `ForumServiceImpl` class in such a way that when the posts are returned, they are filtered out by the rules explained in the last paragraph. To do this, we'll use the `@PostFilter` annotation.

First, we need to add a new voter to the list of voters of our `AccessDecisionManager`. The new voter is of type `org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter` and is shown in Listing 7-20. This voter is needed because even if we'll be using only the `@PostFilter` annotation, Spring Security will try to evaluate the config attributes when it's doing the voting, and it includes a `PreInvocationAttribute` in the list of config attributes that it will evaluate with the value `permitAll`, so the `PreInvocationAuthorizationAdviceVoter` will vote to grant access all the time. The code in Listing 7-20 should be added in the file `applicationContext-security.xml`.

Listing 7-20. PreInvocationAuthorizationAdviceVoter needed to vote on the automatically generated permitAll expression

```
<bean id ="preInvocationAuthorizationAdviceVoter"
         class="org.springframework.security.access.prepost.
         PreInvocationAuthorizationAdviceVoter">
    <constructor-arg>
<bean class="org.springframework.security.access.expression.method.
      ExpressionBasedPreInvocationAdvice"/>
    </constructor-arg>
</bean>
```

Next, we need to change the return type of our `getPosts` method because the filter we'll add works only with instances of `java.util.Collection` or arrays, and as we currently are returning a map from that method, the implementation wouldn't work. We'll now just return a collection of the posts. The new method looks like Listing 7-21. If you are following along with the code, you should change all the classes and files that depend on this method, including the `form.jsp` file.

Listing 7-21. The getPosts method now returns a collection of posts and not the map

```
public Collection<Post> getPosts(){
    return new ArrayList<Post>(postStore.values());
}
```

Next, on top of the method from Listing 7-21, we add the annotation `@PostFilter("hasPermission(filterObject, 'READ')")`, where `filterObject` is each of the objects of the returned collection and `READ` is the permission that matches the `BasePermission.READ` that you saw before. This is where the filtering functionality will get triggered. Next, I'll explain how it works, in the context of an execution scenario. However, first we need to configure a couple of beans manually in the application context to allow the correct evaluation of permission expressions. This is needed because, by default, the SpEL expression evaluator configured in Spring Security will use an `org.springframework.security.access.expression.DenyAllPermissionEvaluator` which, as its name implies, will deny all permission evaluation requests. That configuration is hardcoded in the class `org.springframework.security.access.expression.AbstractSecurityExpressionHandler<T>`, and we need to replace it with a proper evaluator.

Fortunately, Spring Security provides us with the proper evaluator in the form of the class `org.springframework.security.acls.AclPermissionEvaluator`. We need a bit of work to configure it, but it is not that difficult. First, we need to define the bean that will be the new `ExpressionHandler`. We do that in the `applicationContext-security.xml` file by adding the code from Listing 7-22.

Listing 7-22. The ExpressionHandler bean with the correct permission evaluator injected

```
<bean id="customPermissionEvaluator"
        class="org.springframework.security.acls.AclPermissionEvaluator">
        <constructor-arg ref="aclService" />
    </bean>

    <bean id="customExpressionHandler"
        class="org.springframework.security.access.expression.method.
            DefaultMethodSecurityExpressionHandler">
        <property name="permissionEvaluator" ref="customPermissionEvaluator" />
    </bean>
```

Then, in the element `<security:global-method-security>`, add as a child the element `<security:expression-handler ref="customExpressionHandler" />` to make reference to the new expression handler we just defined.

If you are following through in the code, you should restart the application now. After the application is restarted, if you visit the URL http://localhost:8080/forum/ and create a post, you will see that the post won't be shown in the page. This is because you created the post as an `ANONYMOUS` user. Remember that posts now will show only for `ROLE_USER` users. If you log in to the application with the username **car** and the password **scarvarez** and again go to the URL http://localhost:8080/forum/, you will see the post in the page and the delete button. The following paragraphs explain how it all works under the hood.

As you might recall from previous chapters, the core of Spring Security is the concept of the `SecurityInterceptor` and its two personifications: `FilterSecurityInterceptor` and `MethodSecurityInterceptor`. As you probably recall as well, the interceptors work in a pre-process, process, post-process flow. The actual business logic happens in the "process" phase, and both the "pre-process" and "post-process" phases are used for the framework itself to apply all the security concerns, regarding authorization and access control, that are needed to secure the application.

The "pre-process" phase is taken care of mainly by the access decision managers and the access decision voters that decide whether or not access should be allowed to a particular resource (be it a method, a domain object, or a URL). The "post-process" phase is handled by an `AfterInvocationManager` that is called from the `SecurityInterceptor`. By default, an instance of `AfterInvocationProviderManager` is called.

I explained this process before: the `AfterInvocationProviderManager` iterates through a list of `AfterInvocationProvider`, which takes the final decision of whether or not access to a particular domain object instance is allowed. The important part in our current example is that the chain of calls ends in an instance of `ExpressionBasedPostInvocationAdvice` (through the `PostInvocationAdviceProvider`), which checks to see if there is a `postFilter` expression that needs to be handled. If there is, it calls the `DefaultMethodSecurityExpressionHandler` that we defined in Listing 7-22. This handler checks that the returned value from the business method with the `@PostFilter` annotation was indeed a collection or an array (and throws an exception if it wasn't). Then it iterates through this collection, evaluating the SpEL expression on each object and discarding the object to which the evaluation of the expression gives the value `false`. The expression we are using is an ACL expression and, like most of the other expressions, its backing method is defined in the class `SecurityExpressionRoot`. This method (`hasPermission`) uses the permission evaluator we defined in Listing 7-22 to decide if the authentication has the required permissions on the object being passed. This evaluator uses the same suites of classes and helpers that the `AclEntryVoter` uses to decide whether or not to grant access, such as the `AclService` and the method `isGranted` in the ACL interface.

So we know what happened in the example. When we tried to access the list with the anonymous user, the only existing post was discarded from the return elements because the user didn't match the permissions required to read the object. (ACL's `isGranted` method returned `false`.) When we logged in as *car*, *scarvarez*, we acquired the role `ROLE_USER`. This role is indeed allowed to read post objects, as specified by the ACL rules that we created when we first created the post.

Let's see how filtering works in an example with more than one post. This time, we'll create a post with a user with the role ROLE_ADMIN. We'll modify the post creation code so that when an Admin user creates a post, only other ROLE_ADMIN users can read that post. To do that, change the createPost method in the ForumServiceImpl class to look like Listing 7-23. This listing (like many others in the book) takes an approach of preferring convenience over particularly good design. For example, you might argue, and rightly, that the ACL mutation is not part of the core createPost method business functionality, and also that hardcoding role names in the code is not right. Again, I'm doing this to illustrate concepts in a convenient way, without too much abstraction and directly to the point that I'm trying to show. In a real environment, you should try to achieve a good separation of concerns, giving the security concerns their own space (whether using AOP or other methods) and leave the business method to handle, well, business concerns.

Listing 7-23. The createPost method that creates different ACLs depending on the user that is creating the post, along with a needed helper method to check whether the logged-in user has the ROLE_ADMIN role

```
@Transactional
    public void createPost(Post post) {
        Integer id = new Integer(Math.abs(post.hashCode()));
        ObjectIdentity oid = new ObjectIdentityImpl(Post.class, id);
        MutableAcl acl = mutableAclService.createAcl(oid);
        User user = (User)SecurityContextHolder.getContext().getAuthentication().getPrincipal();
        acl.insertAce(0, BasePermission.ADMINISTRATION, new PrincipalSid(
                user.getUsername()), true);
        acl.insertAce(1, BasePermission.DELETE, new GrantedAuthoritySid(
                "ROLE_ADMIN"), true);
        if(isAdminUserLogged()){
           acl.insertAce(2, BasePermission.READ, new GrantedAuthoritySid(
                "ROLE_ADMIN"), true);
        }else{
            acl.insertAce(2, BasePermission.READ, new GrantedAuthoritySid(
                "ROLE_USER"), true);
        }
        mutableAclService.updateAcl(acl);
        post.setId(id);
        postStore.put(id, post);
    }
private booleanisAdminUserLogged() {
    for(GrantedAuthority authority: SecurityContextHolder.getContext().getAuthentication().
    getAuthorities()){
        if(authority.getAuthority().equals("ROLE_ADMIN")){
            return true;
        }
    }
    return false;
}
```

You can see in the listing that we added a conditional saying, basically, that if the logged-in user is an Administrator, the READ permission will be available only to other Administrators (users with the role ROLE_ADMIN). If the logged-in user is not an Administrator, the READ permission will be available to any user with the ROLE_USER role.

I'll now show an execution of this new configuration step by step.

Download from Wow! eBook <www.wowebook.com>

Test Scenario 7-1

To execute this new configuration, I used the following steps:

1. I restarted the application, and then `visited` http://localhost:8080/spring_security_login and logged in with the username **car** and the password **scarvarez**.
2. I then visited the URL http://localhost:8080/forum/ and created a new post with the content *p1*. That created the screen you see in Figure 7-7.



Figure 7-7. *The screen that is generated after a user with the role ROLE_USER creates a post*

3. I visited the URL http://localhost:8080/j_spring_security_logout to log out of the application. Then I visited the URL http://localhost:8080/spring_security_login and logged in with the username **mon**" and password **scarvarez**.
4. I visited the URL http://localhost:8080/forum/ and created a post with the content *p2*. I got to Figure 7-8.

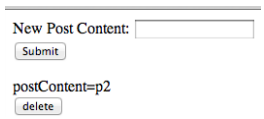


Figure 7-8. *The sceen that is generated after an admin user creates a post*

5. I visited the URL http://localhost:8080/j_spring_security_logout to log out of the application. Then I visited the URL http://localhost:8080/spring_security_login and logged in with the username **bea** and the password **scarvarez**.
6. I visited the URL http://localhost:8080/forum/, and I was able to see both posts as Figure 7-9 shows.



Figure 7-9. *Posts for a user with both the ROLE_USER and ROLE_ADMIN roles*

7. I visited the URL http://localhost:8080/j_spring_security_logout to log out of the application. Then I visited the URL http://localhost:8080/spring_security_login and logged in with the username **car** and the password **scarvarez**.
8. I visited the URL http://localhost:8080/forum/, and I can only see the Post post1 as it is the only one accessible for standard users. The view is exactly the same as Figure 7-7.

Spring Security ACL support also offers the `@PreFilter` annotation which, as you can imagine, works in the pre-process phase of the method security interception process rather than the way that the `@PostFilter` annotation works. As I said before, the pre-processing activities are taken care of by `AccessDecisionVoter` implementations. In the case of the `@PreFilter` annotation, it is taken care of by the `org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter` which, in turn, uses an instance of `DefaultMethodSecurityExpressionHandler` as in the previous case. As in the case of `@PostFilter`, `@PreFilter` annotated methods are expected to receive a collection. In this case, however, arrays are not supported.

I just showed you a simple walkthrough of how this ACLfiltering functionality works in practice. You can see that by providing only the `@PostFilter` annotation with the `hasPermission` expression (which, as you know, internally calls a method in the SpEL context), we are instructing the framework what to do with the returned values of the method. This process that we explained first, and demonstrated later in the small test case, can be summarized with Figure 7-10, Figure 7-11, and Figure 7-12.

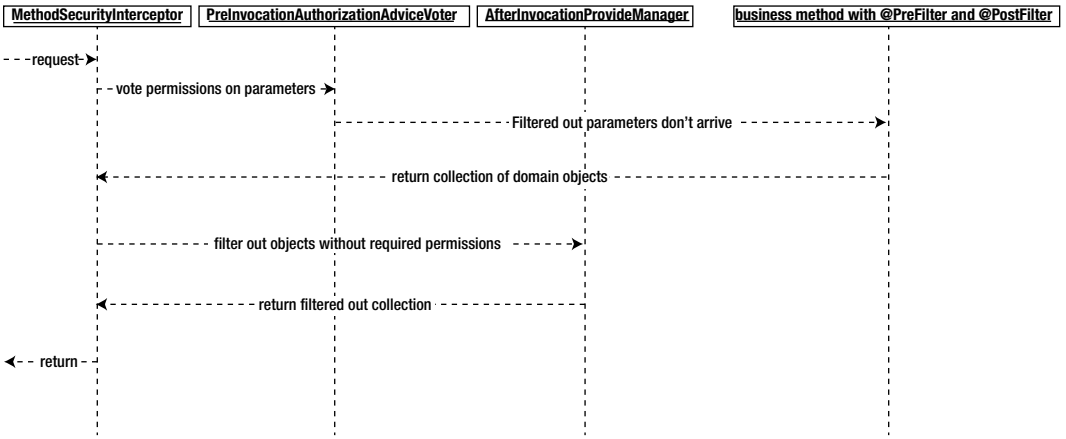


Figure 7-10. *A simplified sequence diagram of what happens when a method with @PreFilter and @PostFilter annotations is invoked. The diagram shows how parameters and return collections of domain objects are filtered out on the way in and out of the method*

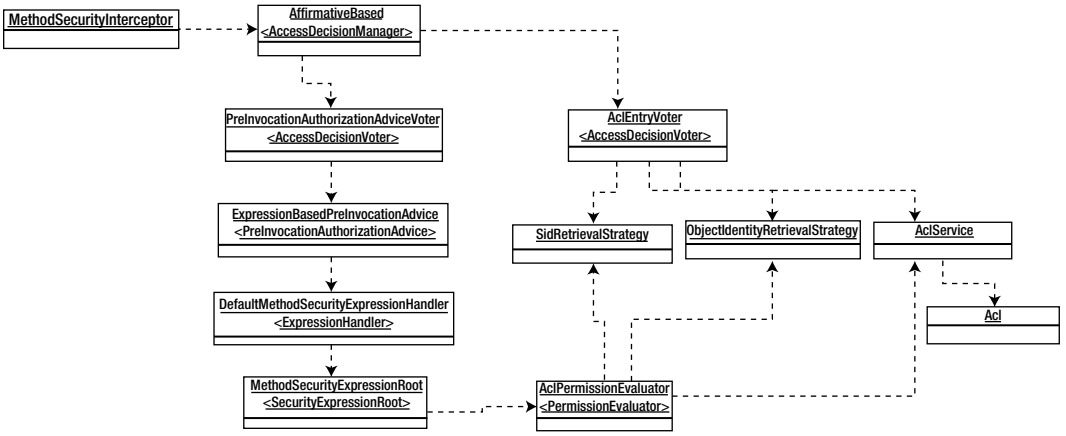


Figure 7-11. *A diagram showing the main classes and interfaces in the pre-processing phase of a method interception with ACL-based security*

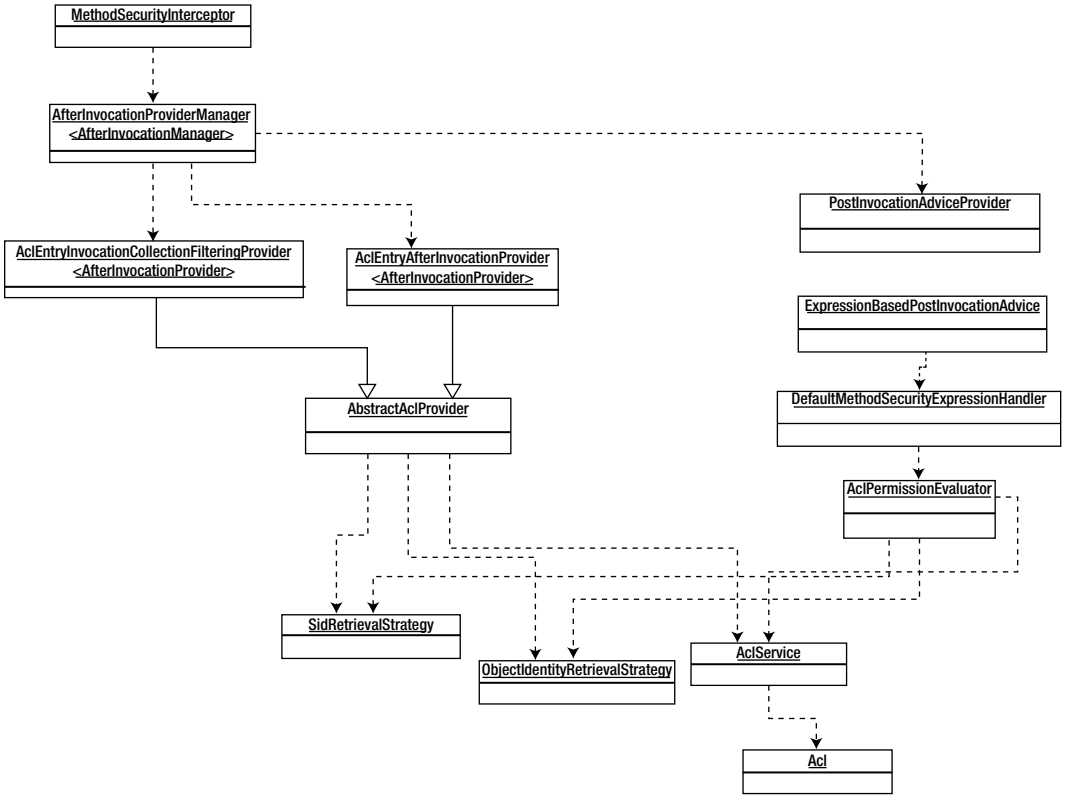


Figure 7-12. *A diagram showing the classes that participate in the post-processing phase of a method interception with ACL-based security*

`@PreFilter` invocations are handled by the `PreInvocationAuthorizationAdviceVoter`, while `@Secured` annotations with ACL config attributes are handled by the `AclEntryVoter`. You can see in the diagram that, at the end of the processing, both paths reach the essential elements in the `SidRetrievalStrategy`, `ObjectIdentityRetrievalStrategy`, and `AclService`. In fact, the `AclPermissionEvaluator` and the `AclEntryVoter` perform very similar functions.

Again, we can see two branches. The branch on the left, where you can see the `AclEntryAfterInvocationProvider` and `AclEntryAfterInvocationCollectionFilteringProvider`, is the branch that doesn't support SpEL expressions and is active when using `@Secured` and a static name for a config attribute, something like `ACL_ALLOW_READ` or whatever, as we saw a couple of sections back. The right branch goes through the section that supports SpEL and both `@PostFilter` and `@PostAuthorize` annotations. You can see that these two branches map almost one to one to the branches in the pre-processing phase. Also, you can see that there is a high level of reuse in the system and that many classes are present in both the pre-processing and post-processing phases.

I covered two different ways to handle ACL security, with the two branches I talked about in the previous paragraph. The one that uses `@Secured` is older and doesn't support SpEL expressions; the one using `@PreFilter`, `@PreAuthorize`, `@PostAuthorize`, and `@PostFilter` has existed since version 3.0 of Spring Security. They are newer options that support the use of SpEL expressions, so you will probably favor these over the other ones in your day-to-day use.

Securing the View Layer with ACLs

Another option you have for using ACLs to secure applications is to use the view-layer JSP tags to filter out domain objects for users who don't have the proper permissions to see them. I talked a bit about these tags in the web security chapter, but I intentionally left the ACL tags for more thorough treatment here. To use them in an example, let's continue with our code from the previous section, but let's make a couple of changes. First, in the `ForumServiceImpl` class, comment out the `@PostFilter` annotation in the `getPosts` method so that it doesn't filter out anything anymore. Then change the `form.jsp` file to look like Listing 7-24.

Listing 7-24. The form.jsp with taglib ACL security applied for filtering domain objects

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="security"
    uri="http://www.springframework.org/security/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Posts</title>
</head>
<body>
<form method="post" action="/forum/post">
        New Post Content: <input type="text" name="postContent"/><br/>
        <input type="submit"/>
    </form>
    <c:forEach items="${posts}" var="post">
    <security:accesscontrollist  domainObject="${post}" hasPermission="READ">
    <form method="post" action="/forum/post/delete">
            ${post.content} <br />
            <input type="hidden" value="${post.id}" name="postId"/>
            <input type="submit" value="delete"/>
        </form>
    </security:accesscontrollist>
    </c:forEach>
</body>
</html>
```

This listing shows how simply you can secure domain objects on the view layer. In fact, it does a very similar job (in view terms) to what you get when you use the `@PostFilter` annotation.

The `<accesscontrollist>` tag depends on the configured `org.springframework.security.access.PermissionEvaluator` implementation that exists in the application context configuration. It will work only if there is only one bean of this type configured in the application context, because it will try to retrieve the bean by type. As you might recall, we have a configured `PermissionEvaluator` in our configuration of concrete type `org.springframework.security.acls.AclPermissionEvaluator`.

Now, if you restart the application and again execute the steps we defined in Test Scenario 7-1, you should have exactly the same behavior from a presentation layer point of view as you had before when you first executed that scenario. The way it works is simple. Every time the tag is evaluated (in each iteration over the posts collection), the tag handler calls the `PermissionEvaluator`'s `hasPermission` method, which returns a Boolean indicating whether or not permission has been granted. If permission is granted, the body of the tag is evaluated and rendered. If permission is not granted, the body of the tag is skipped.

The Cost of ACLs

The example we have been working on shows that working with ACLs imposes a good deal of overhead in terms of the operations and logic you need to implement on top of your standard business operations. It's clear that storing a post is affected by the extra work that is necessary to store the corresponding ACL for that object, and retrieving posts is affected by the need to filter out certain elements of the collection based on the permissions.

This brief section will go inside the core class of the ACL Spring Security support to see how it works internally and how it might affect your application. This class is the `org.springframework.security.acls.jdbc.JdbcMutableAclService`.

`JdbcMutableAclService` is the class that deals with all the input and output to the database for everything regarding the ACL database schema. It's configured by default to work with `HSQLDB`, which is the database we have been using in our examples.

This class has different SQLs for the different things you do when you interact with the ACL system—like inserts into the different tables, deleting ACLs, updating values and, of course, selecting from the different tables. All of these operations are done using Spring Core's `JdbcTemplate` support.

Another important class is `BasicLookupStrategy`, the default implementation of `LookupStrategy` used in the framework to look up ACLs.

The first interesting operation you see is the method `readAclsById`, which exists in `JdbcMutableAclService` and delegates to a method of the same name in `BasicLookupStrategy`. This method tries to retrieve the requested Acl from the cache. If it finds the ACL in cache the method will return this found ACL; if the ACL is not found in the cache, the implementation will query the database with a somewhat complex query with four joins, as Listing 7-25 shows. This query can get a bit more complex because the ACLs can be retrieved in batches of up to 50 elements (which is the default but is configurable by setting the property `batchSize` in the class `BasicLookupStrategy`) and, for each of these elements, the `where` clause adds an `or` operator to the query. The result from this query is then stored in the cache, which then makes these ACLs available so that if another method calls `readAclsById` requesting one of the cached elements, the database doesn't need to be hit.

When you create a new post, the `readAclsById` is called. Also, another select query is performed to retrieve the primary key of the `objectIdentity` represented by the domain object. And, of course, an "`insert into acl_object_identity`" is performed.

Also, in the same `createPost` method in `ForumServiceImpl`, when you call the `updateAcl` method, a "`delete from acl_entry`" is performed, followed by a batch "`insert into` acl_entry", followed by an "`update acl_object_identity`". Then all the caching entries for that Object Identity are cleared and a new call to the `readAclsById` is executed which will query the database for the up to date information. You can see how the cost of deleting a post has increased considerably with the use of ACLs.

Deleting `Post` objects is also more costly because now it involves deleting ACE entries, deleting object identities, and clearing the cache for the relevant objects.

This section is not meant to scare you away from using ACLs. I simply want to make you aware that there is an extra cost (apart from the complexity of using it) you should take into consideration when creating your applications with the use of ACLs in mind. The more domain objects you have, the more ACL entries you will have as well. In a big application, you might be talking about millions of entries in the ACL support tables.

Listing 7-25. An acl retrieving query that you can find in the class org.springframework.security.acls.jdbc.BasicLookupStrategy

```
select acl_object_identity.object_id_identity, acl_entry.ace_order,  acl_object_identity.id as
acl_id,
acl_object_identity.parent_object, acl_object_identity.entries_inheriting, acl_entry.id as
ace_id, acl_entry.mask,
acl_entry.granting,  acl_entry.audit_success, acl_entry.audit_failure,  acl_sid.principal as
ace_principal,
acl_sid.sid as ace_sid,  acli_sid.principal as acl_principal, acli_sid.sid as acl_sid,
acl_class.class from
acl_object_identity left join acl_sid acli_sid on acli_sid.id = acl_object_identity.owner_sid
left join acl_class on acl_class.id = acl_object_identity.object_id_class
left join acl_entry on acl_object_identity.id = acl_entry.acl_object_identity
left join acl_sid on acl_entry.sid = acl_sid.id
where ( (acl_object_identity.object_id_identity = ? and acl_class.class = ?))
order by acl_object_identity.object_id_identity asc, acl_entry.ace_order asc
```

Summary

In this chapter, I explained in detail how to use Spring Security's support for ACLs. I showed you how to configure support for ACL in the Spring configuration file and how to use the `@Secured`, `@PreAuthorize`, `@PreFilter`, `@PostAuthorize`, and `@PostFilter` annotations to implement domain object–specific security. I also introduced some internal aspects of the framework and the main classes that are involved in its ACL functionality.

We examined different ways to make sure that secured domain objects don't show up in the presentation layer for a user who doesn't have appropriate permissions. You saw that this is achievable either with SpEL expressions at the `@PostFilter` business level or with the ACL Spring Security tag library directly in your JSP files. I also gave a quick overview of the different SQLs that are used by the ACL framework and how they might impact your application.

CHAPTER 8

Customizing and Extending Spring Security

Spring Security is a very extendable and customizable framework. This is primarily because the framework is built using object-oriented principles and design practices so that it is open for extension and closed for modification. In the previous chapter, you saw one of the major extension points in Spring Security—namely, the pluggability of different authentication providers. This chapter covers some other extension points in the framework that you can take advantage of to extend Spring Security's functionality or to modify or customize functionality that doesn't work exactly the way you need in your applications.

I also briefly cover the Spring Security Extensions project (`http://static.springsource.org/spring-security/site/extensions.html`), an environment you can use to create extension modules for the core Spring Security project.

The next section defines what I consider to be some of the major extension points in Spring Security and describes how to use them to add or modify behavior in your security solution.

Spring Security Extension Points

Spring Security offers a comprehensive set of extension points that can be customized (or completely overridden) with your own implementations and still leverage the core of the framework. Some of the extension points are evident, while some others are a bit more subtle and, in some cases, not even intended. However, because the framework is so flexible, you can take advantage of that flexibility to tweak its configuration to fit your intentions.

Plug into the Spring Security Event System

Spring Security supports an event model that is built on top of Spring Framework's own event model. You can use Spring's event model to develop applications that can listen to different events that happen within the framework and act accordingly.

I won't explain in any depth why an event model is such a powerful programming practice to have at your disposal. Instead, I'll just point out one big advantage: it allows you to decouple your applications, because in general the event producer or producers and the event consumer or consumers don't need to know anything about each other in order to operate correctly. In theory (and, indeed, in practice for events in general, although not for Spring events), you can have a completely heterogeneous application where you can write and evolve each module at its own pace without affecting other parts, and then integrate them all together through the exclusive use of events.

All Spring events should extend from the abstract class `org.springframework.context.ApplicationEvent`, and Spring Security's own events are no exception.

One of the main concrete implementations of the `ApplicationEvent` abstract class is `org.springframework.context.event.ApplicationContextEvent`, which itself serves as the parent class of a series of events that involve the life cycle of the application context (`ContextClosedEvent`, `ContextRefreshedEvent`, `ContextStartedEvent`, `ContextStoppedEvent`).

To register your application to be notified of Spring events, you need one (or more) of the beans defined in your application to implement the interface `org.springframework.context.ApplicationListener<E extends ApplicationEvent>`. This interface defines a single method, `void onApplicationEvent(E event)`, that you can use to listen to a particular type of event in the application.

Publishing events is equally easy. You only need to define a bean in your Spring application context that implements the interface `org.springframework.context.ApplicationEventPublisherAware`, which again defines only one method:

```
void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher)
```

This method is called automatically by Spring when the application starts up, and an instance of `ApplicationEventPublisher` (an implementation of it because it is an interface) is passed in. The instance of `ApplicationEventPublisher` that is passed in is normally the `ApplicationContext` instance itself that contains the application.

The default implementation of `ApplicationEventPublisher`'s `publishEvent` method (which lives in the class `AbstractApplicationContext`) delegates the publishing of the events to an implementation of `ApplicationEventMulticaster`—the only current implementation of which is `org.springframework.context.event.SimpleApplicationEventMulticaster`.

Broadcasting an event to all interested listeners is also straightforward. You simply need to create an instance of one of the implementing classes of `ApplicationEvent` (any subclass of it will do as an event) and then call the `ApplicationEventPublisher`'s `publishEvent(ApplicationEvent event)` method, passing your `ApplicationEvent` instance to it. Spring then takes care of ensuring that all the listeners registered for that particular event are notified of the event publication. By default, all listeners are invoked on the same thread as the publisher; however, you also could configure an `org.springframework.core.task.TaskExecutor` (which is an interface, so you could use an implementation class like `org.springframework.core.task.SimpleAsyncTaskExecutor`) to call the listeners in different threads. You will use this when you configure the listeners in our examples.

Spring Security comes with its own suite of `ApplicationEvent` implementations, so you can hook into different points of the security life cycle in an unobtrusive and decoupled way. The `ApplicationEvent` implementations that Spring Security provides are categorized as `Authorization`, `Authentication`, or `javax.servlet.http.Session` types, and they have descriptive names that hint what they do and where they are published. Here, I will give a concrete explanation of them and when they are published within the framework.

Events in Spring work as shown in Figure 8-1.

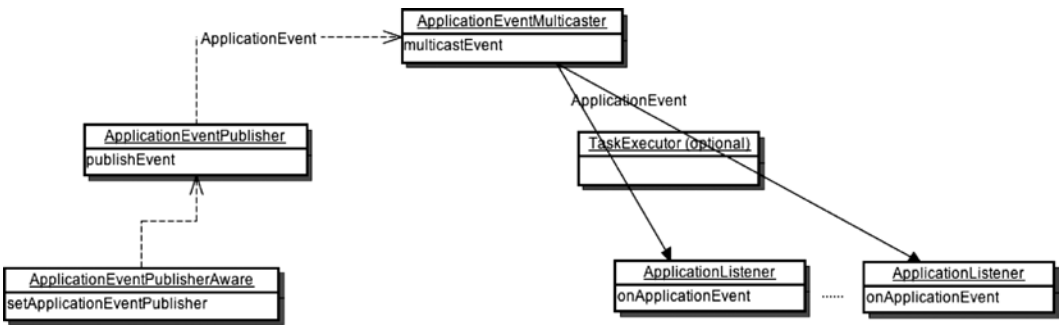


Figure 8-1. *Spring event mechanism. Spring Security has its own set of ApplicationEvent implementations*

Authorization-Related Events

These are events related to the different phases that an authorization process can go through—for example, informing a user when an authorization has failed. The events under this category are the following:

- **`org.springframework.security.access.event.AbstractAuthorizationEvent`** This is the parent class of all the other authorization events. It doesn't really add any functionality on top of `ApplicationEvent`. It is more like a marker identifying all its subclasses as authorization events.
- **`org.springframework.security.access.event.AuthenticationCredentialsNotFoundEvent`** This event is used to indicate that the `Authentication` object could not be obtained from the configured `SecurityContext`. This event is broadcasted from the `AbstractSecurityInterceptor`'s `beforeInvocation` method after it decides that an invocation should have security but the interceptor doesn't find the required `Authentication` object in the security context. Simply put, this event is executed if the condition `SecurityContextHolder.getContext().getAuthentication() == null` is `true`.
- **`org.springframework.security.access.event.AuthorizationFailureEvent`** This event indicates that the `Authentication` object's principal was not allowed access to a secured object because it doesn't have the required permissions to access it. This event is published by the `AbstractInterceptor`'s `beforeInvocation` and `afterInvocation` methods when it catches an `AccessDeniedException`. `AccessDeniedException` can be thrown by the `org.springframework.security.access.AccessDecisionManager`'s `decide` method in the pre-processing phase of the interceptor, and by the `org.springframework.security.access.intercept.AfterInvocationProviderManager`'s `decide` method in the post-process part of the interceptor.
- **`org.springframework.security.access.event.AuthorizedEvent`** This event is published after access to a secured object has been granted and before actually invoking the secured object. This event is not published by default, and if you want it to be published, you need to set the property `publishAuthorizationSuccess` to `true` in the security interceptor.
- **`org.springframework.security.access.event.PublicInvocationEvent`** This event is published by the `AbstractSecurityInterceptor`'s `beforeInvocation` method whenever a secured object receives an invocation to a nonsecured entry point (that is, a method without `ConfigAttribute` configured). In the case of method security, this means that when an object's security proxy is invoked, if the particular invoked method is not configured with security metadata, it is then handled as a nonsecured public call. However, instead of simply invoking the method, an event is broadcasted just before proceeding to inform to any listener interested in this fact that this (a public invocation) has happened. This could point to a case when you actually need to secure the endpoint and, thanks to the event, you will be notified that you haven't done so. After the event is published, no more pre-processing or post-processing is executed for this invocation on the interceptor. If the property `rejectPublicInvocations` is set to `true` in the interceptor (the property is part of the `AbstractSecurityInterceptor` class), the event will not be published, and instead an `IllegalArgumentException` will be thrown saying that the particular invocation cannot be done without the `ConfigAttribute`(s) configured. This means that a configuration error needs to be taken care of.

Figure 8-2 shows these event classes and interfaces in UML (Unified Modeling Language) form.

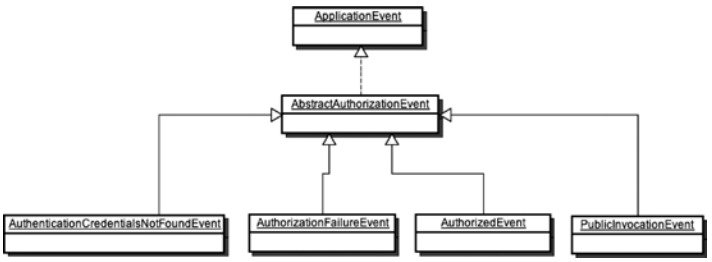


Figure 8-2. *Authorization-related events*

Authentication-Related Events

These events are related to the authentication process in the application and the different phases this process goes through, such as informing interested listeners of disabled accounts and expired credentials. Authentication events are published by an implementation of `org.springframework.security.authentication .AuthenticationEventPublisher`, which is invoked by the `org.springframework.security.authentication .ProviderManager` class. By default, the configured `AuthenticationEventPublisher` in the `ProviderManager` is an instance of `NullEventPublisher`, which is a private static class defined inside the `ProviderManager` class itself and which doesn't publish any events. There exists a default implementation of the interface `AuthenticationEventPublisher` that you can use if you are configuring the beans yourself, which is `org.springframework.security.authentication.DefaultAuthenticationEventPublisher`. This instance is configured by default for the web-layer security when using the `<http>` element.

The `DefaultAuthenticationEventPublisher` works in the following way: It has only two public methods that it implements from the `AuthenticationEventPublisher` interface. These methods are `publishAuthenticationSuccess` and `publishAuthenticationFailure`. These are the possible two scenarios when attempting authentication; however, in the case of authentication failure, there can be many reasons for it, as you can see from the amount of `AuthenticationFailureXXXEvent instances` that I'll show you next. In the case of authentication failures, the `DefaultAuthenticationEventPublisher` will receive a call to the `publishAuthenticationFailure` method, and then it will query a mapping between the exception that was thrown when authentication was denied (like `UsernameNotFoundException`, for example) and the event that corresponds to such an exception (like `AuthenticationFailureBadCredentialsEvent`). Then an instance of this event will be created by reflection and will be published.

It is possible to add more mapping into the default mappings between exceptions and events using the method provided by the `DefaultAuthenticationEventPublisher`: `setAdditionalExceptionMappings`.

The authentication-related events currently in the framework are the following:

- **`org.springframework.security.authentication.event.AbstractAuthenticationEvent`** This is the parent class of all other authentication-related events. It doesn't introduce any particular functionality, and it serves the basic function of classifying authentication events.
- **`org.springframework.security.authentication.event.AbstractAuthenticationFailureEvent`** This is the parent class of all the authentication failure events. It extends `AbstractAuthenticationEvent` but adds a constructor that takes in the exception that was thrown when the authentication failure happened.
- **`org.springframework.security.authentication.event.AuthenticationFailureBadCredentialsEvent`** This event is published when a `BadCredentialsException` is thrown by the system during authentication. This exception is thrown by the different `AuthenticationProvider` implementations when a check for the credentials of an `Authentication` object is not valid. The event is also published when a `UsernameNotFoundException` is thrown. This is thrown normally by `UserDetailsService` implementations when they can't find a user corresponding to the passed username and simply propagated by the `AuthenticationProvider`.

Download from Wow! eBook <www.wowebook.com>

- **`org.springframework.security.authentication.event.AuthenticationFailureCredentialsExpiredEvent`** This event is published when a `CredentialsExpiredException` is thrown. This exception is thrown, for example, by an implementation of `AbstractUserDetailsAuthenticationProvider` when the `UserDetails` object representing the user returns `false` from the method `isCredentialsNotExpired`.
- **`org.springframework.security.authentication.event.AuthenticationFailureDisabledEvent`** This event is published when a `DisabledException` is thrown. This exception is thrown if the user account that is trying to log in has been disabled. It is used by `AbstractUserDetailsAuthenticationProvider` and also `AccountStatusUserDetailsChecker` implementations evaluating if the `UserDetails.isEnabled` method returns `false`.
- **`org.springframework.security.authentication.event.AuthenticationFailureExpiredEvent`** This event is published by the `ProviderManager` when an `AccountExpiredException` is thrown. This exception is thrown following the same logic as the previous case, but this time the `UserDetails.isAccountNonExpired` method is the one that is called. If it returns `false`, the exception is thrown.
- **`org.springframework.security.authentication.event.AuthenticationFailureLockedEvent`** This event is published when a `LockedException` is thrown. This exception is thrown in the same places as the previous one (in pre-authentication checking scenarios) and is thrown when the `UserDetails` representing the user returns `false` from its method `isAccountNonLocked`.
- **`org.springframework.security.authentication.event.AuthenticationFailureProviderNotFoundEvent`** This event is different than the previous ones in that it is not related directly with the user attempting to log in. Instead, this event is published when a `ProviderNotFoundException` is thrown. This exception is thrown by the `ProviderManager` itself if none of the configured `AuthenticationProviders` configured in the `ProviderManager` are able to handle the authentication request.
- **`org.springframework.security.authentication.event.AuthenticationFailureServiceExceptionEvent`** This event gets published when an `AuthenticationServiceException` is thrown. This exception can be thrown by different parts of the system (for example, `UsernamePasswordAuthenticationFilter` and `DaoAuthenticationProvider`), and it is generally used for communicating that the authentication could not be processed due to some sort of system error—for example, if the user repository cannot be accessed.
- **`org.springframework.security.authentication.event.AuthenticationSuccessEvent`** This event is simply published by the `DefaultAuthenticationEventPublisher` when its `publishAuthenticationSuccess` method is called.
- **`org.springframework.security.authentication.event.InteractiveAuthenticationSuccessEvent`** This event is published directly by different filters in the web-layer security part of the framework, and not by the `DefaultAuthenticationEventPublisher`. It is published whenever a successful authentication is achieved by any of the filters that actively try authentication, like the `UsernamePasswordAuthenticationFilter` or the `RememberMeAuthenticationFilter`.
- **`org.springframework.security.authentication.jaas.event.JaasAuthenticationEvent`** This is the parent event of both `JaasAuthenticationFailedEvent` and `JaasAuthenticationSuccessEvent`, and they are published by the `JaasAuthenticationProvider`.

Figure 8-3 shows the mentioned event classes in a UML class diagram.

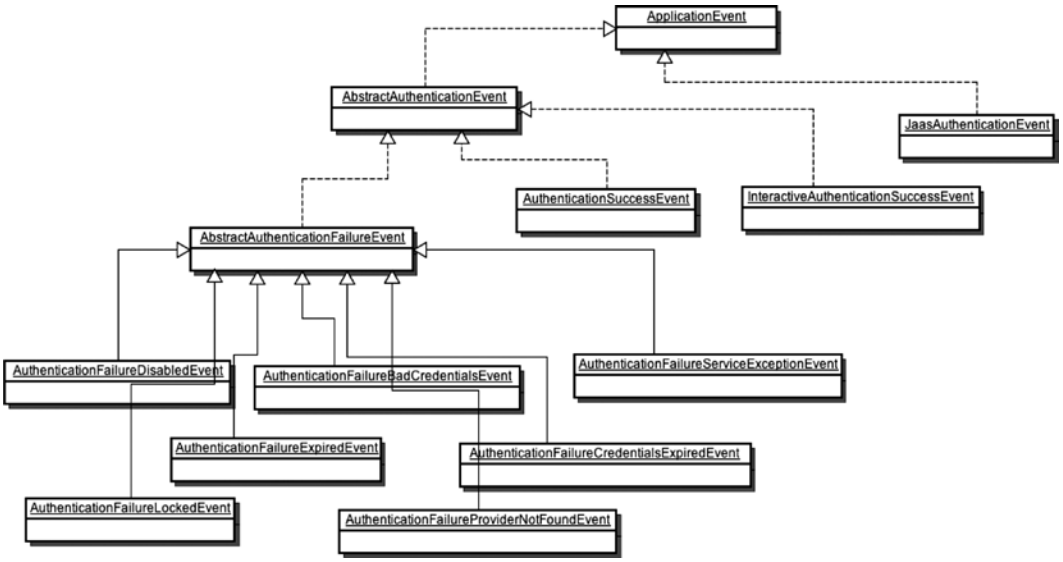


Figure 8-3. *Authentication-related event classes*

Session-Related Events

These events (`org.springframework.security.core.session.SessionCreationEvent` and `org.springframework.security.core.session.SessionDestroyedEvent`), which extend directly from `ApplicationEvent`, are related to the user session life cycle. They are both published (actually, their concrete implementing subclasses `HttpSessionCreatedEvent` and `HttpSessionDestroyedEvent`) by an instance of `HttpSessionEventPublisher`, an implementation of the standard servlet interface `HttpSessionListener`, which allows the `HttpSessionEventPublisher` to create session life-cycle listeners. The implementation needs to be referenced in the `web.xml` file like any other `HttpSessionListener`. The two session-related events map one-to-one to the two methods defined by the `HttpSessionListener` interface. These methods are

```
void sessionCreated(HttpSessionEvent se)
void sessionDestroyed(HttpSessionEvent se)
```

This is all the theory you need to know about publishing and handling events in Spring Security. Next you will see a very simple example where you put this knowledge into practice to add behavior to your application based on listening to events.

This example will listen to only one type of event, but the configuration needed to listen to more types is exactly the same. You implement the same interface but type it differently in the generic type.

All you need do to start listening for events is implement the `ApplicationListener` interface and configure the implementing bean in the Spring application context. Listing 8-1 shows a simple implementation that logs `AuthenticationFailureBadCredentialsEvent events`.

Listing 8-1. ApplicationListener That Listens to AuthenticationFailureBadCredentialsEvent and Logs It

```
package com.apress.pss.security;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.context.ApplicationListener;
import org.springframework.security.authentication.event.AuthenticationFailureBadCredentialsEvent;

public class LoggerBadCredentialsEvents implements
ApplicationListener<AuthenticationFailureBadCredentialsEvent>{

    private static Log LOG = LogFactory.getLog(LoggerBadCredentialsEvents.class);
    public void onApplicationEvent(
            AuthenticationFailureBadCredentialsEvent event) {
        LOG.warn("An attempt to login with bad credentials was made with username "+
event.getAuthentication().getName());

    }

}
```

The implementation of the listener is very straightforward. You need to implement only the interface and type it with the class of the event that you want to listen to. Then if you define a bean instance of the class `LoggerBadCredentialsEvents` in the application context, it will automatically be wired into the Spring Framework event-handling system, and every time an `AuthenticationFailureBadCredentialsEvent` is published, this listener will be notified of it.

To define any other event handler, you do the same but type the handler class by the correct event that you want to listen to. Go ahead and try it yourself. It should be simple enough to test this functionality for different event types.

Your Own AuthenticationProvider and UserDetailsService

You have seen in the previous chapter that Spring comes equipped with quite a few authentication options to adjust to a lot of different application requirements that you might have in your application. Looking at how these different authentication providers are set up in your application and the nice way they are contained in their own "modules," you could think that you should be able to use your own authentication provider. The truth is, of course, that you can, and here I will show you how with a simple example.

First of all, let's review how the authentication providers work in your application.

The main authentication entry point in Spring Security is the `AuthenticationManager` interface—in particular, the `ProviderManager` implementation. `ProviderManager`'s main functionality is to iterate through a list of `AuthenticationProvider` implementations that are configured on it until one of them is able to authenticate the user (wrapped in one of the available `Authentication` implementation objects) or, in fact, until discovering that none of them can, at which point it throws an exception.

The standard way you saw in previous chapters for defining the `ProviderManager` and the `AuthenticationProvider` is the one shown in Listing 8-2. (Although the class names and the names of the elements in the XML file don't exactly match, the Spring Security namespace XML parsing mechanism takes care of matching the combination shown of the `<authentication-manager>` and `<authentication-provider>` elements to the classes mentioned before.) When you use the namespace to define an authentication manager and an authentication provider as shown in the listing, the framework instantiates two `ProviderManager` objects and sets one as the parent of the other. This scheme is used by the `ProviderManager` which is able to establish a hierarchy of authentication managers. The child `AuthenticationManager` (in form of the `ProviderManager` class) is queried first for authentication. If it doesn't

resolve the authentication request, it checks to see if it has a configured parent manager that it can query; if it does, it calls its authenticate method. This parent–child relationship is managed by Spring at startup time.

In the parent–child relationship of the `ProviderManager` instances, one of them has defined in the `AuthenticationProvider` list a `DaoAuthenticationProvider` instance, coming from the `<authentication-provider>` element, and the other has an `AnonymousAuthenticationProvider` configured, and this one is defined when the `<http>` element is used and the common filters are being defined. So these are the two providers that are used by default when you use the common definition from Listing 8-2.

Defining your own custom authentication provider is simple, as you just need to implement the interface `org.springframework.security.authentication.AuthenticationProvider`, which defines only two methods:

```
Authentication authenticate(Authentication authentication)
boolean supports(Class<?> authentication)
```

But implementing the interface is not the only option. You could easily extend `AbstractUserDetailsAuthenticationProvider` (the way that `DaoAuthenticationProvider` works) if the implementation you want to create depends on the `UserDetails` abstraction for authentication. The main method in the `AuthenticationProvider` is the `public Authentication authenticate(Authentication authentication)` method. This method, as you can see, receives an authentication and returns an authentication. The important difference between the two authentication objects (the one received and the one returned) is that the `Authentication` object it returns will return `true` in the method `isAuthenticated` (if authentication is successful, of course), indicating that a fully authenticated object is now in existence, while the `Authentication` object received in the method will have this method returning `false`. This is the logical way to work, as this method in an `Authentication` object (`isAuthenticated`) and its return value are what conceptually differentiate an `Authentication` object that is fully authenticated from one that is used only to wrap the user details before actually applying the authentication logic; or, indeed, any `Authentication` object that hasn't yet been fully authenticated and verified.

Sometimes, you might not need to implement a whole `AuthenticationProvider`, but instead you just might need to change the place from where the `UserDetails` is obtained. That is the case, for example, when using the `InMemoryUserDetailsManager` for getting `UserDetails` stored in memory or using `JdbcUserDetailsManager` when getting `UserDetails` stored in a relational database. Of course, you can create a new `UserDetailsService` to retrieve this `UserDetails` from some other source, and that is something I'll show you in the upcoming examples. Figure 8-4 illustrates the relationship between the `AuthenticationProvider` (actually, the `AbstractUserDetailsAuthenticationProvider` implementation) and the `UserDetailsService`.

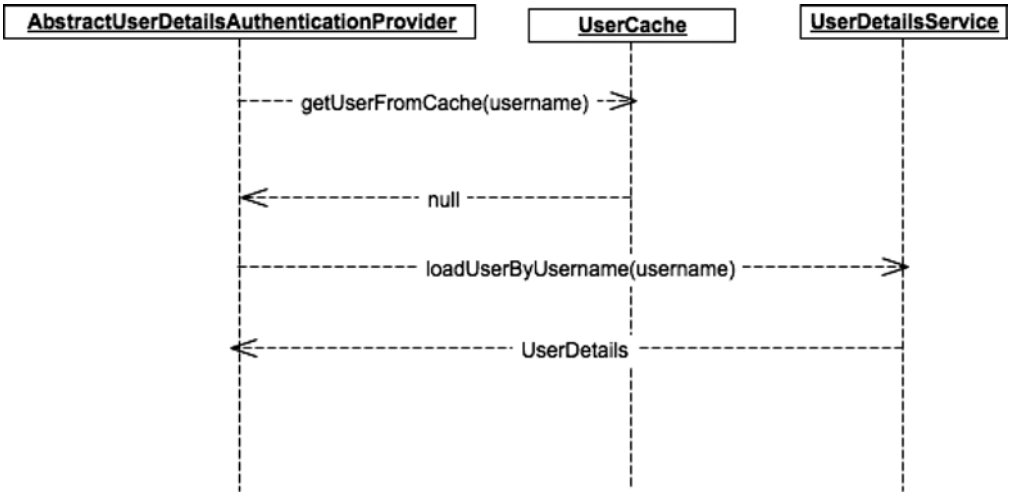


Figure 8-4. *AbstractUserDetailsAuthenticationProvider and UserDetailsService relationship*

In Figure 8-4, both AbstractAuthenticationProvider and UserDetailsService can be replaced by custom implementations. The UserCache, by default, uses a NullUserCache implementation, which is a no-op cache. It can also be replaced easily, but I don't cover that here. The idea of the cache is to keep the UserDetails objects cached in case they need to be retrieved again. The authenticate method in the AbstractUserDetailsAuthenticationProvider will look first for UserDetails in this cache; if the authenticate method finds the UserDetails in the cache, it will use that UserDetails to try the authentication. If the authenticate method doesn't find UserDetails in the cache, it will query the configured UserDetailsService for the user and then store it on the cache for subsequent requests.

Listing 8-2. Common Definition of an AuthenticationManager and AuthenticationProvider. To Be Included in applicationContext-security.xml

```
<security:authentication-manager>
    <security:authentication-provider>
        <security:user-service>
            <security:user authorities="ROLE_XX" name="x" password="xx" />
        </security:user-service>
    </security:authentication-provider>
</security:authentication-manager>
```

The first example I'll show is how to create your own UserDetails service and plug it into the AuthenticationProvider. For this, you will simply use the DaoAuthenticationProvider, whose functionality is precisely to delegate the retrieval of UserDetails to a UserDetailsService and authenticate the retrieved user. In the example, I will also introduce the option to exchange password encoders in order to use different algorithms to cypher the passwords, instead of storing them in plain text. The first thing we'll do is get the code from Chapter 2, as you will use it as a base for all your examples in this chapter.

Some things need to change or get discarded. In my case, I changed the pom.xml a little bit to use a different project and artifact name. I also removed all the files from the "discard" package and configured the <intercept-url> element to intercept all URLs. (You should know how to do this by now. If you do not, next is a quick step-by-step reminder on how to configure the applications from this chapter.) You can do the same with your configuration or simply start from scratch, remembering to copy the required dependencies in your pom.xml.

For you to have a running application and to not leave you guessing exactly what parts of Chapter 2 to grab here, I give you the step-by-step guide to get all the needed application setup. I won't explain the steps, as they have already been covered; I will just give a quick list of the steps to help you follow along with the examples. In future sections when a new project is started to test some new functionality, you can refer to this part to create a new project from scratch:

1. Execute the following:

   ```
   mvn archetype:generate -DgroupId=com.apress.pss -DartifactId=example1
   -DarchetypeArtifactId=maven-archetype-webapp
   ```

2. In the pom.xml file, add the dependencies from Listing 8-3.
3. In the pom.xml file, add the plugin from Listing 8-4 to the build section.
4. Create the applicationContext-security.xml from Listing 8-5, and put it in the WEB-INF folder of the application.
5. Create the web.xml from Listing 8-6 in the application.
6. Create the simple servlet from Listing 8-7 in the package com.apress.pss.servlets.
7. If you run mvn clean install, you should have a running application with enabled Spring Security.

Listing 8-3. Dependencies for the Examples Bootstrap

```
<dependency>
     <groupId>javax.servlet</groupId>
     <artifactId>javax.servlet-api</artifactId>
     <version>3.0.1</version>
</dependency>
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-core</artifactId>
     <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-config</artifactId>
     <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
     <groupId>org.springframework.security</groupId>
     <artifactId>spring-security-web</artifactId>
     <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
     <groupId>commons-logging</groupId>
     <artifactId>commons-logging</artifactId>
     <version>1.1.1</version>
</dependency>
<dependency>
     <groupId>commons-codec</groupId>
     <artifactId>commons-codec</artifactId>
     <version>1.3</version>
</dependency>
<dependency>
     <groupId>org.springframework</groupId>
     <artifactId>spring-webmvc</artifactId>
     <version>3.1.2.RELEASE</version>
</dependency>
```

Listing 8-4. Jetty Plugin in the Build Section of the pom.xml

```
<build>
        <plugins>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>8.1.1.v20120215</version>
                <configuration>
                    <connectors>
                        <connector implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
                            <port>8080</port>
                            <maxIdleTime>60000</maxIdleTime>
                        </connector>
```

```
                    </connectors>
                </configuration>
            </plugin>
        </plugins>
        <finalName>example1</finalName>
</build>
```

Listing 8-5. applicationContext-security.xml That Serves as a Base for Some of the Examples

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true">
        <security:intercept-url pattern="/*"
            access="ROLE_SCARVAREZ_MEMBER" />
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user name="car" authorities="ROLE_SCARVAREZ_MEMBER" password="scarvarez"/>
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans>
```

Listing 8-6. Basic web.xml with Enabled Spring Security

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>

    <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
         /WEB-INF/applicationContext-security.xml
        </param-value>
    </context-param>

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

```
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Listing 8-7. Simple "Hello World" Servlet

```
package com.apress.pss.servlets;

import java.io.IOException;

import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

@WebServlet(urlPatterns={"/hello"})
public class HelloWorldServlet extends HttpServlet {
    private static final long serialVersionUID = 2218168052197231866L;

    @Override
    public void doGet(HttpServletRequest request, HttpServletResponse response){
        try {
            response.getWriter().write("Hello World");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

You now have a running application. Remember that security is currently configured, as shown in Listing 8-5. Let's create a new `UserDetailsService` implementation that will look for users in a MongoDB database. Like many other examples in this book, the implementation I'll show you will be somewhat trivial, but it will serve to show you the steps you need to implement the `UserDetailsService`. It is very simple, so let's get going.

First of all, let's add the MongoDB dependencies to our `pom.xml`. You will be using Spring Data to set up and use MongoDB. It is not really required, but because you are working with Spring it seems like a good idea. According to the Spring Data for MongoDB (http://www.springsource.org/spring-data/mongodb) website, it "aims to provide a familiar and consistent Spring-based programming model for new datastores while retaining store-specific features and capabilities." So you have a Spring-based application and are using MongoDB, which is a good fit for it. The dependency you need to add to the `pom.xml` file is shown in Listing 8-8.

Next, of course, you need MongoDB itself. You can download it from its official site at http://www.mongodb.org/downloads. After you download it (and assuming you are working on either Linux or Mac OSX), you can simply unpack the file somewhere and it is ready to run. To run the MongoDBserver, simply go to the bin directory of the directory you just unpacked and run the file `./mongod`. If you have Windows, you should find easy-to-follow instructions in the MongoDB website.

■ **Note** MongoDB is a very popular document-based database. It is part of the many NoSQL solutions that have become so popular in the recent past. It is a very powerful and flexible tool. Its main power comes from the fact (at least from my perspective) that it combines a very scalable storage solution with a very intuitive document model built on top of known technologies (like JSON and JavaScript in the command-line interface). And it does this without forgetting about one of the best things of the SQL world, which is the flexibility given by the availability to execute dynamic queries, allowing you to query your database using very varied criteria and filtering options. This contrasts with, for example, key-value storage solutions that allow searching only by the key. As I said, MongoDB is a Document store, where the Documents are structured as JSON. MongoDB stores its Documents in collections of documents called *collections*. I won't go into explaining MongoDB any further, except for what is needed to create and run the example. If you are interested in learning more about it, there is a lot of bibliography online that can help you out, starting with the project's website and, in particular, this link to current books on the topic: http://www.mongodb.org/display/DOCS/Books.

Listing 8-8. spring-data-mongodb Dependency

```
<dependency>
 <groupId>org.springframework.data</groupId>
 <artifactId>spring-data-mongodb</artifactId>
 <version>1.1.0.RELEASE</version>
</dependency>
```

You now have the MongoDB server running and the needed dependencies in your project. Next let's write the `UserDetailsService` implementation. To do this, you need to implement only one method:

```
UserDetails loadUserByUsername(String username)
```

Implementing it looks like Listing 8-9.

Listing 8-9. MongoUserDetailsService. Implementation of UserDetailsService That Uses MongoDB to Retrieve the UserDetails

```
package com.apress.pss.security;

import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;

public class MongoUserDetailsService implements UserDetailsService {

    private MongoTemplate mongoTemplate;

    public MongoUserDetailsService(MongoTemplate mongoTemplate){
        this.mongoTemplate = mongoTemplate;
    }
```

```
    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        UserDetails user = mongoTemplate.findOne(new Query(Criteria.where("username").is(username)),
                User.class, "users");
        if (user == null){
            throw new UsernameNotFoundException("Username "+username+ " not found.");
        }
        return user;
    }

}
```

That's all there is to implementing the `UserDetailsService`. You can see how nice the Spring Data MongoDB solution looks, and you should be able to understand to some degree what the code is doing. The `MongoUserDetailsService` is getting a `MongoTemplate` reference at construction time, which it then sets on an instance variable. The `loadUserByUsername` method uses this template to retrieve the user from the Mongo database. As you can see, the `loadUserByUsername` method calls the findOne method in the mongoTemplate to try and retrieve the user by username, telling it (in the second parameter to findOne) the class of the object it expects its result to be unmarshalled to and the Mongo collection where it is trying to find the specified document. The `User` class, which is an implementation of `UserDetails`, is the class of the objects that we expect to get back from the method call.

That is all specified by the following simple method call:

```
UserDetails user = mongoTemplate.findOne(new
Query(Criteria.where("username").is(username)),User.class, "users");
```

Its three parameters define the information I just mentioned:

- A query that specifies the information you want to get using filtering.
- The class of the object you expect to be returned from the query. Internally, the `MongoTemplate` and some helpers will take care of the conversion from the native Mongo objects to this object type. It will use a set of mapping converters to help in this conversion.
- The collection that you want to query with the template. Remember that a collection is simply a, well, collection of documents. You normally store similar documents in the same collection, and if you are familiar with relational databases, you can think of collections conceptually as tables from the relational world, although they are not the same.

You can also see in the code listing that you are throwing a `UsernameNotFoundException` in case the MongoDB query you are executing returns `null`. With this, you remain consistent with the way other `UserDetailsService` implementations work and what the `AuthenticationProvider` expects. Actually, returning `null` from `UserDetailsService` is a "contract violation" for the interface, and you will be informed of this if you accidentally do it.

The next thing you need to do is configure the application so that it is actually aware that it will be using MongoDB. Then you must configure the new `UserDetails` service in the `AuthenticationProvider`. First you create a new configuration file in the `WEB-INF` folder named `applicationContext-mongodb.xml` with the content from Listing 8-10.

Listing 8-10. applicationContext-mongodb.xml Defines the Configuration of the MongoDB Database Where You Will Store the Users

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


Download from Wow! eBook <www.wowebook.com>

```
xmlns:mongo="http://www.springframework.org/schema/data/mongo"
    xsi:schemaLocation="http://www.springframework.org/schema/data/mongo
          http://www.springframework.org/schema/data/mongo/spring-mongo-1.0.xsd
          http://www.springframework.org/schema/beans
          http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

    <mongo:mongo host="localhost" port="27017" />
    <mongo:db-factory dbname="example1" />

    <mongo:mapping-converter>
        <mongo:custom-converters>
            <mongo:converter>
                <bean class="com.apress.pss.security.UserReadConverter" />
            </mongo:converter>
            <mongo:converter>
                <bean class="com.apress.pss.security.UserWriteConverter" />
            </mongo:converter>
        </mongo:custom-converters>
    </mongo:mapping-converter>

    <bean id="mongoTemplate" class="org.springframework.data.mongodb.core.MongoTemplate">
        <constructor-arg name="mongoDbFactory" ref="mongoDbFactory" />
        <constructor-arg name="mongoConverter" ref="mappingConverter"/>
    </bean>
</beans>
```

You can see that this file is using a `<mongo:>` namespace. This is part of the Spring Data project's MongoDB support. Here you are simply defining where your database is listening for connections (`localhost:27017`), specifying what the name of your database (example1) is, and creating the `MongoTemplate` instance you will inject into your `UserDetails` service.

More importantly, you can see that you are defining two converter instances inside a `<mapping-converter>` element and then using this converter in the `mongoConverter` property of the `MongoTemplate`. You can use these two converters to map the documents from the Mongo database to and from the `User` objects you are using. These converters are defined in Listings 8-11 and 8-12.

Listing 8-11. UserReadConverter Allows You to Convert MongoDB Objects into Your Domain User Object

```
package com.apress.pss.security;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;

import org.springframework.core.convert.converter.Converter;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;

import com.mongodb.BasicDBList;
import com.mongodb.DBObject;
```

```
public class UserReadConverter implements Converter<DBObject, User> {

    public User convert(DBObject source) {
        return new User((String) source.get("username"),
                 (String) source.get("password"),
                convertAuthoritiesReading(source));
    }

    private Collection<? extends GrantedAuthority> convertAuthoritiesReading(
            DBObject source) {
        List<?> stringAuthorities = (BasicDBList) source.get("authorities");
        List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
        for (Object stringAuth : stringAuthorities) {
            authorities.add(new SimpleGrantedAuthority((String)stringAuth));
        }
        return authorities;
    }
}
```

Listing 8-12. UserWriteConverter Allows You to Convert Your Domain User Object into MongoDB Objects

```
package com.apress.pss.security;

import org.springframework.core.convert.converter.Converter;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.User;

import com.mongodb.BasicDBObject;
import com.mongodb.DBObject;

public class UserWriteConverter implements Converter<User,DBObject> {

    public DBObject convert(User source) {
        DBObject dbObject =  new BasicDBObject();
        dbObject.put("username", source.getUsername());
        dbObject.put("password", source.getPassword());
        dbObject.put("authorities", authoritiesAsStringArray(source));
        return dbObject;
    }

    private Object authoritiesAsStringArray(User source) {
        String[] authorities = new String[source.getAuthorities().size()];
        int i = 0;
        for(GrantedAuthority auth: source.getAuthorities()){
            authorities[i]=auth.getAuthority();
            i++;
        }
        return authorities;
    }

}
```

The converters in both Listing 8-11 and Listing 8-12 implement the same `Converter` interface, but they invert the order of conversion of the types in the type parameters of the class. They basically do inverse operations. `UserReadConverter` makes sure you can read the MongoDB document into a `User` object. The relevant data is extracted from the document and is transformed when needed (as with the GrantedAuthority) to set it as properties of the object, and of course it calls the correct constructor on the User class.

Listing 8-12 is the opposite. It takes a `User` object and extracts the values from its properties to store them on the MongoDB collection creating a Mongo `DBObject, which ultimately will represent a Mongo document`. These converters are automatically invoked with the definition you wrote in Listing 8-10.

Now you have to change the `applicationContext-security.xml` file in a couple of ways. The first thing that needs to be done is to import the new `applicationContext-mongodb.xml` file. The second thing is to define the new `UserDetailsService` bean and make it part of the security flow by injecting it in the `AuthenticationProvider`. In the end, your `applicationContext-security.xml` file should look something like Listing 8-13.

Listing 8-13. applicationContext-security.xml with the New MongoUserDetailsService Configured

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <import resource="applicationContext-mongodb.xml"/>
    <security:http auto-config="true">
        <security:intercept-url pattern="/*" access="ROLE_SCARVAREZ_MEMBER" />
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider user-service-ref="mongoUserDetailsService"/>
    </security:authentication-manager>
    <bean id="mongoUserDetailsService" class="com.apress.pss.security.MongoUserDetailsService">
     <constructor-arg ref="mongoTemplate"/>
    </bean>
</beans>
```

Although everything is configured right now, you still don't have any users created in your users collection in your `example1` Mongo database. There is no 100 percent standard way of doing this from Spring Security's point of view; however, since version 2.0 it offers the interface `UserDetailsManager`, which extends `UserDetailsService` with a set of CRUD methods that allow you to create users, update them, delete them, and change their passwords. This interface has JDBC, LDAP, and InMemory implementations, but, of course, there isn't a MongoDB one. What you will do is modify your `MongoUserDetailsService` to implement this interface instead of the simple `UserDetailsService`. Then you will create a simple program to store a user using this service. The new `MongoUserDetailsService` is shown in Listing 8-14, and the simple main program to insert a `User` is shown in Listing 8-15.

Listing 8-14. MongoUserDetailsService, Which Allows You to Store a UserDetails Instance in the Mongo Database

```
package com.apress.pss.security;

import org.springframework.data.mongodb.core.MongoTemplate;
import org.springframework.data.mongodb.core.query.Criteria;
import org.springframework.data.mongodb.core.query.Query;
import org.springframework.security.core.userdetails.User;
```

```
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.provisioning.UserDetailsManager;

public class MongoUserDetailsService implements UserDetailsManager {

    private MongoTemplate mongoTemplate;

    public MongoUserDetailsService(MongoTemplate mongoTemplate){
        this.mongoTemplate = mongoTemplate;
    }

    public UserDetails loadUserByUsername(String username)
            throws UsernameNotFoundException {
        UserDetails user = mongoTemplate.findOne(new Query(Criteria.where("username").is(username)),
                User.class, "users");
        if (user == null){
            throw new UsernameNotFoundException("Username "+username+ " not found.");
        }
        return user;
    }

    public void createUser(UserDetails user) {
        mongoTemplate.insert(user, "users");

    }

    public void updateUser(UserDetails user) {
        throw new UnsupportedOperationException();
    }

    public void deleteUser(String username) {
        throw new UnsupportedOperationException();
    }

    public void changePassword(String oldPassword, String newPassword) {
        throw new UnsupportedOperationException();
    }

    public boolean userExists(String username) {
        throw new UnsupportedOperationException();
    }

}
```

You can see that Listing 8-14 added the method `createUser` to the class, which simply uses the `MongoTemplate` to store the user in one line. The listing also has a lot of new methods, which you are not going to implement but which are required by the `UserDetailsManager` interface.

Listing 8-15. Small Main Class to Insert One User in the MongoDB, Leveraging the MongoUserDetailsService

```
package com.apress.pss;

import java.util.Arrays;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.provisioning.UserDetailsManager;

public class UserInserter {
    public static void main(String[] args){
        ApplicationContext context =
new FileSystemXmlApplicationContext("//....../WEB-INF/applicationContext-security.xml");
        UserDetailsManager userDetailsManager = context.getBean(UserDetailsManager.class);
        GrantedAuthority[] authorities =
new GrantedAuthority[] {new SimpleGrantedAuthority("ROLE_SCARVAREZ_MEMBER")};
        UserDetails user = new User("car", "scarvarez", Arrays.asList(authorities));
        userDetailsManager.createUser(user);
    }
}
```

The code in Listing 8-15 is very straightforward. You are creating a new user with username "car", password "scarvarez", and the role `ROLE_SCARVAREZ_MEMBER`. Then you are using your new `UserDetailsManager` (the `MongoUserDetailsService`) implementation to store the user in the Mongo database. This is so that you have a user for your tests. Run this main class as you would normally run any Java class with a main method to store the user. After running this class, you should have the user created in the collection. You can check this out by completing the following steps:

1. Go to the directory where MongoDB was installed.
2. Go inside the `bin` folder.
3. Execute the `./mongo` command.
4. Execute the commands shown in Figure 8-5.

```
Carlos-MacBook-Air:bin cscarioni$ ./mongo
MongoDB shell version: 2.2.0
connecting to: test
> use example1
switched to db example1
> db.users.find()
{ "_id" : ObjectId("50b3d11b300434dfb673d73e"), "username" : "car", "password" : "scarvarez", "author
ities" : [ "ROLE_SCARVAREZ_MEMBER" ] }
```

Figure 8-5. *MongoDB looking at the newly created user*

The application is ready to go; start it with `mvn jetty:run` and give it a go. If you visit the URL `http://localhost:8080/hello`, you will be prompted for the username and password. By introducing "car" and "scarvarez", respectively, and navigating to the URL `http://localhost:8080/hello`, you should be able to access the "Hello World" message page, which is shown in Figure 8-6.

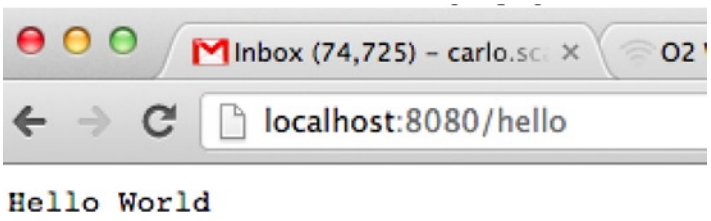


Figure 8-6. *The "Hello World" page*

So you have created a new `UserDetailsService` implementation backed up by MongoDB, and you are using it in the `AuthenticationProvider` to retrieve the logged-in user.

Password Encryption

One thing you might have noticed in this example and all the previous examples is that you are storing and retrieving passwords in plain text. Spring Security offers an abstraction to encrypt passwords in the form of the interface `org.springframework.security.authentication.encoding.PasswordEncoder` in the core framework. This interface is currently implemented by a series of classes that offer everything from plain-text encoders (such as the default one implemented in class `PlaintextPasswordEncoder`, which doesn't do any encryption in the password) to hashing passwords that use recognized encryption algorithms (like `Md5PasswordEncoder` and `ShaPasswordEncoder`). To use one of these implementations with your current `AuthenticationProvider`, you need to define a `<password-encoder>` element in the configuration file as a child of the `<authentication-provider>` element. For example, if you want to use Hash-256 as the digest algorithm for your passwords, you make the `<authentication-provider>` element look like Listing 8-16. The element `<password-encoder>` currently allows you to define any of seven options in the "hash" attribute: `md4`, `md5`, `sha`, `sha-256`, `{sha}`, `{ssha}`, and `plaintext`. The `{sha}` and `{ssha}` options are used for Lightweight Directory Access Protocol (LDAP) in the `LdapShaPasswordEncoder`. Another way of configuring the encoder is to use the `ref` attribute instead of the `hash` and specify a reference to a bean that holds a `PasswordEncoder` instance. This, of course, can be an encoder implementation you create yourself. I will do this to be able to reference the same encoder in the writing part of the `UserInserter`. In doing so, I will be able to store the encoded value of the password, ensuring that I'm using exactly the same algorithm implementation as the one that will be used when logging in to the application. So the passwords should match if they are the same. To do this, I simply define a bean somewhere in the file, as Listing 8-17 shows, and then use the ID of that bean in the `ref` attribute of the `<password-encoder>` element.

Of course, as I mentioned in the previous paragraph, you now need to configure the insertion of the user to use the encoding algorithm as well so that Spring Security is able to match the password when retrieving the user from the database. To do this, you will use the password encoder defined and retrieved from your `UserInserter` class before storing the password. For that, you make the main method from the `UserInserter` class look like Listing 8-18. (You need to add the import `org.springframework.security.authentication.encoding.PasswordEncoder;` to the class from this listing).

Listing 8-16. <authentication-provider> with a sha-256 Password Encoder Configured

```
<security:authentication-provider user-service-ref="mongoUserDetailsService">
 <security:password-encoder hash="sha-256"/>
</security:authentication-provider>
```

Listing 8-17. sha-256 Individual Bean That Is Then Referenced in the <password-encoder> Element

```
<bean id="passwordEncoder"
class="org.springframework.security.authentication.encoding.ShaPasswordEncoder">
<constructor-arg value="256" />
</bean>
```

Listing 8-18. User Inserter Main Method Now Hashes the Password Before Inserting It

```
public static void main(String[] args){
        ApplicationContext context =
new FileSystemXmlApplicationContext("//......./WEB-INF/applicationContext-security.xml");
        UserDetailsManager userDetailsManager = context.getBean(UserDetailsManager.class);
        GrantedAuthority[] authorities =
new GrantedAuthority[] {new SimpleGrantedAuthority("ROLE_SCARVAREZ_MEMBER")};
        PasswordEncoder encoder = context.getBean(PasswordEncoder.class);
        UserDetails user =
new User("car", encoder.encodePassword("scarvarez",null), Arrays.asList(authorities));
        userDetailsManager.createUser(user);
    }
```

If you execute this code now, you will insert the user with the password hashed with the SHA-256 algorithm. You can check that by querying your MongoDB database, as explained in Figure 8-5 and its preceding paragraph. After doing this, you can restart your application and use "car" and "scarvarez" to log in as you did before. You should be able to log in without any problems, with no apparent difference from a user interface point of view. However, now, under the hood your passwords are not stored in plain text anywhere in the application, so they cannot be compromised easily.

As I said at the beginning of the section, sometimes you might want to replace the whole `AuthenticationProvider` implementation instead of simply the `UserDetailsService`. You could also create another `AuthenticationProvider` implementation to combine with the existing ones because, if you remember, the `AuthenticationManager` default implementation (`ProviderManager`) is able to iterate through a list of `AuthenticationProvider` instances, and also, it is able to relate to other `AuthenticationManager` instances in a parent–child relationship. There are many different implementations of `AuthenticationProvider`, as you saw in Chapter 7, including support for LDAP, JAAS and others. You could implement your own `AuthenticationProvider` if you need to. You simply need to define an `<authentication-provider>` element as a child of the `<authentication-manager>` element and use its `ref` attribute to reference your implementation bean.

Most of the `AuthenticationProvider` implementations decide if they can handle a particular authentication request by consulting the type of the `Authentication` object that is passed to the authenticate method, using the `supports` method to check. I explained this process in detail in Chapter 7, so I won't go into it any further here.

New Voters in AccessDecisionManager

As I explained before, `AccessDecisionManager`'s default implementations (`AffirmativeBased`, `UnanimousBased`, and `ConsensusBased`) work by querying a set of configured `AccessDecisionVoters` to allow or deny access to a particular resource. It is fairly straightforward to implement your own `AccessDecisionVoter`, and that is what I will show you here. I will start again from the simple code I explained in the first example of this chapter (refer to that part of the

chapter for a step-by-step guide to setting up your project) and work from there. Again, do any modifications required to the `pom.xml` file if you would like to name the application differently.

Let's go directly to business here. You will create an `AccessDecisionVoter` that will vote on attributes of the form `USERNAME_XX`, where it will grant access to any user whose username is `XX`. (You could implement this solution using a SpEL expression, but we want to show the use of custom voters, so this example will do.) The first thing you will do is create the voter implementation itself. It should be something like Listing 8-19.

Listing 8-19. UsernameVoter Is a Custom AccessDecisionVoter

```
package com.apress.pss.security;

import java.util.Collection;

import org.springframework.security.access.AccessDecisionVoter;
import org.springframework.security.access.ConfigAttribute;
import org.springframework.security.core.Authentication;

public class UsernameVoter implements AccessDecisionVoter<Object> {

    public boolean supports(ConfigAttribute attribute) {
        if ((attribute.getAttribute() != null)
                && attribute.getAttribute().startsWith("USERNAME_")) {
            return true;
        } else {
            return false;
        }

    }

    public boolean supports(Class<?> clazz) {
        return true;
    }

    public int vote(Authentication authentication, Object object,
            Collection<ConfigAttribute> attributes) {
        int result = ACCESS_ABSTAIN;
        String username = authentication.getName();
        for (ConfigAttribute attribute : attributes) {
            if (this.supports(attribute)) {
                result = ACCESS_DENIED;
                String stringedAttribute = attribute.getAttribute();
                String shortedAttribute = attribute.getAttribute().
substring(stringedAttribute.indexOf("_")+1, stringedAttribute.length());
                if (shortedAttribute.equalsIgnoreCase(username)) {
                    return ACCESS_GRANTED;
                }
            }
        }
        return result;
    }

}
```

This code specifies that this voter supports any `ConfigAttribute` that starts with the string `USERNAME_`. Then, when it is time to vote on the authentication request, it will compare the username of the requesting `Authentication` with the `ConfigAttribute` substring (after the `USERNAME_ prefix`) to see if the user should be allowed access to the secured resource. If it matches, access is granted; if it doesn't, access is denied.

The `applicationContext-security.xml` configuration file should look like Listing 8-20.

Listing 8-20. applicationContext-security.xml with Custom Voter Configured

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true" access-decision-manager-ref="accessDecisionManager">
        <security:intercept-url pattern="/*" access="USERNAME_CAR" />
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user authorities="NOT_USED" name="car" password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
    <bean id="accessDecisionManager"
class="org.springframework.security.access.vote.AffirmativeBased">
        <constructor-arg>
            <list>
            <bean class="com.apress.pss.security.UsernameVoter"/>
            </list>
        </constructor-arg>
    </bean>
</beans>
```

In the file from Listing 8-20, you can see that we have put NOT_USED as one of the authorities of the created user. This is because, as you know, the `UsernameVoter` doesn't read this property; instead, it simply queries the `username` property of the principal of the `Authentication` object. Basically, this is an arbitrary string here as `authorities` is a required attribute of the `<user-element>`, but you know you are not using it in our example so you choose a descriptive name to point out this fact.

This is all the configuration you need to configure a custom `AccessDecisionVoter`. If you restart the application with this new configuration and log in with the username "car" and the password "scarvarez", you should be able to access the `http://localhost:8080/hello` URL without problems.

Nonvoter AccessDecisionManager Implementations

The previous sections show you that the current `AccessDecisionManager` implementations are based on the `AccessDecisionVoter` concept. Basically, they work by iterating through a list of voters to which they delegate the intermediate decision of whether or not to allow access. Depending on the result of the voters and the particular implementation of the `AccessDecisionManager`, they allow or deny access to a resource.

You can implement your own `AccessDecisionManager` that doesn't need to follow the voter approach. Again, this is not going to be a very common piece of the framework to customize, as the voter system is very flexible and well designed. However, you also can extend the existing voter model with a custom `AccessDecisionManager` that handles the execution and priority of voters differently. Actually, extracted directly from Spring Security's own documentation, you find this line: "It is possible to implement a custom `AccessDecisionManager` that tallies votes differently. For example, votes from a particular `AccessDecisionVoter` might receive additional weighting, whilst a deny vote from a particular voter might have a veto effect."

So let's create an `AccessDecisionManager` that allows access if the logged-in user has a granted authority with a name that matches the name of the URL being invoked with removed slash marks. Listing 8-21 shows the `AccessDecisionManager` implementation. The class is a toy, so it would break if method security interception were enabled.

Listing 8-21. AccessDecisionManager Implementation That Grants Access if the Logged-in User Has a Granted Authority with the Same Name as the URL Being Invoked

```
package com.apress.pss.security;

import java.util.Collection;

import org.springframework.security.access.AccessDecisionManager;
import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.access.ConfigAttribute;
import org.springframework.security.authentication.InsufficientAuthenticationException;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.web.FilterInvocation;

public class URLAccessDecisionManager implements AccessDecisionManager{

    public void decide(Authentication authentication, Object object,
            Collection<ConfigAttribute> configAttributes)
            throws AccessDeniedException, InsufficientAuthenticationException {
        FilterInvocation invocation = (FilterInvocation)object;
        String url = invocation.getRequestUrl().replaceAll("/", "");
        boolean granted = false;
        for(GrantedAuthority authority:authentication.getAuthorities()){
            if (authority.getAuthority().equals(url)){
                granted = true;
                break;
            }
        }
        if(!granted){
        throw new AccessDeniedException("Access denied");
        }
    }

    public boolean supports(ConfigAttribute attribute) {
        return true;
    }
```

Download from Wow! eBook <www.wowebook.com>

```
    public boolean supports(Class<?> clazz) {
        return true;
    }

}
```

You can see the code is simply retrieving the URL from the `FilterInvocation` object and comparing it to the list of authorities that the authenticated user has. If they match, access is granted; if they don't, access is denied. Figure 8-7 shows graphically what this simple `AccessDecisionManager` implementation does. An example configuration file to go with this `AccessDecisionManager` can be found in Listing 8-22.

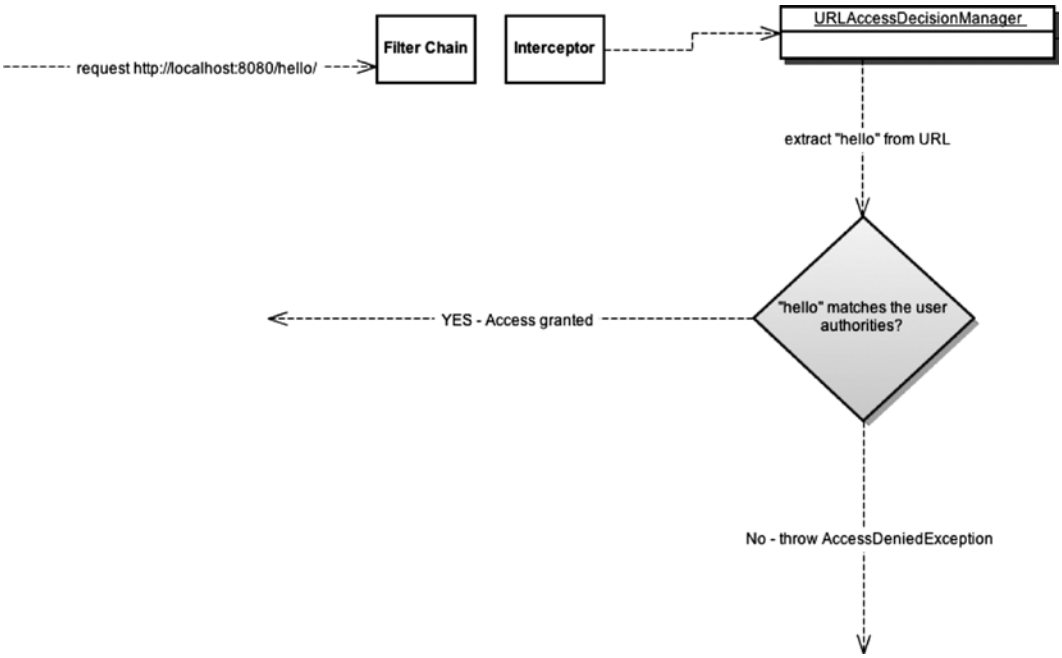


Figure 8-7. *URLAccessDecisionManager Grants Access if the User Has the Simplified URL Name as a Granted Authority*

Listing 8-22. Configuration File applicationContext-security.xml That Shows How to Use a Custom AccessDecisionManager Implementation in Your Application

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true" access-decision-manager-ref="URLAccessDecisionManager">
        <security:intercept-url pattern="/*" access="ROLE_ROS" />
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider>
```

```
            <security:user-service>
                <!-- <security:user authorities="ROLE_USER" name="car" password="scarvarez" /> -->
                <!-- <security:user authorities="ROLE_ADMIN,ROLE_USER" name="mon"
password="scarvarez" /> -->
                <security:user authorities="hello" name="car" password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
    <bean id="URLAccessDecisionManager" class="com.apress.pss.security.URLAccessDecisionManager"/>
</beans>
```

In the previous listing, note the attribute `access-decision-manager-ref` in the `<http>` element. Also note that I added the bean corresponding to the new `AccessDecisionManager`.

New Expression Root and SpEL

In other chapters, I covered the work of SpEL in the context of Spring Security. Note that SpEL is a general functionality provided by the core Spring Framework, which is available to other projects covered by the Spring suite of projects, including Spring Security.

I showed you before how to add some extra functionality to the SpEL processing by creating a different root with an extra method. Specifically, you can look at Chapter 5, where I introduced SpEL and customized its support by defining new expressions.

Non-JDBC AclService

The current support for access control lists (ACLs) in Spring Security is related to the use of relational databases, JDBC, and Spring's `JdbcTemplate`. There is no reason why you couldn't implement ACLs on top of a different storage solution, and there could be many reasons why you would want to do that. For example, it could be because the JDBC solution is too slow, or it can even be that your application doesn't support relational databases at all. As I explained in the ACL chapter, most of the ACL functionality resides in a couple of classes and services.

It is not trivial to implement, but the fact that the SQL support is encapsulated in just a few classes makes the change very focused.

The two main interfaces you need to implement to support a different solution than SQL are the `AclService` (and the `MutableAclService`) and `LookupStrategy`. These interfaces are currently implemented by the `BasicLookupStrategy` and by `JdbcMutableAclService` and `JdbcAclService`.

You could think about implementing these services in many different ways, maybe using some of the NoSQL solutions that are so popular now, like MongoDB, CouchDB, or similar.

Custom Security Filter

As you know, one of the main elements of Spring Security is the web-layer security support leveraged significantly by the Servlet Filter chain that it provides. You can create your own filters and make them part of the security filter chain that processes the request that arrives in the application. That's what we'll do in this next example.

Let's use a simple example here. Say that you want to allow normal users to log in only through a Firefox web browser, while administrator users can log in with any browser they want. This is, of course, an artificial example to show you how to use a custom filter, but there might be some legitimate reason why you would like to do something like this. Maybe your application is verified just for one browser, but you want to allow administrator users to try other browsers to test them out and verify them later for widespread use.

Let's get directly to business and create the filter. You will be using the `User-Agent http` header to identify the browser that is making the request. Listing 8-23 shows the filter implementation.

Listing 8-23. UserAgentFilter That Denies Access to Authenticated Users Without the Role ROLE_ADMIN If They Don't Use Firefox to Access the Application

```
package com.apress.pss.security;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
import javax.servlet.http.HttpServletRequest;

import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.authentication.AuthenticationTrustResolver;
import org.springframework.security.authentication.AuthenticationTrustResolverImpl;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.web.filter.GenericFilterBean;

public class UserAgentFilter extends GenericFilterBean{

    private static final String ADMIN_ROLE = "ROLE_ADMIN";
    private static final String FIREFOX_AGENT_CONTAINS = " Firefox ";
    private AuthenticationTrustResolver authenticationTrustResolver = new
AuthenticationTrustResolverImpl();

    public void doFilter(ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        if(((HttpServletRequest)request).getHeader("User-Agent").
contains(FIREFOX_AGENT_CONTAINS)){
            chain.doFilter(request, response);
        }else{
            processRequestWhenNotExpectedUserAgent(request,response,chain);
        }
    }

    private void processRequestWhenNotExpectedUserAgent(
ServletRequest request, ServletResponse response,
            FilterChain chain) throws IOException, ServletException {
        if(isUserAnAdmin() || isUserAnonymous()){
            chain.doFilter(request, response);
        }else{
            throw new AccessDeniedException(
"The browser you are using is not supported for your user role. Use Firefox instead");
        }
    }

    private boolean isUserAnonymous() {
        return SecurityContextHolder.getContext().
        getAuthentication() != null &&
```

```
                authenticationTrustResolver.isAnonymous(
                 SecurityContextHolder.getContext().getAuthentication());    }

    private boolean isUserAnAdmin() {
        if(SecurityContextHolder.getContext().getAuthentication() != null){
            for(GrantedAuthority ga :
SecurityContextHolder.getContext().getAuthentication().getAuthorities()){
                if(ga.getAuthority().equals(ADMIN_ROLE)){
                    return true;
                }
             }
        }
        return false;
    }

}
```

In the listing, you can see how the functionality is implemented. You have hard-coded in a couple of constants. One is for the administrator role that is allowed access from any browser. In the other constant (`FIREFOX_AGENT_CONTAINS`), you specify what string to look for in the `User-Agent` header to grant access to logged-in users. Anonymous authenticated users are allowed to use any browser because in the false theory of my application, the anonymous accessible parts (if any) are working for any browser. You can also see that the filter extends from the class `GenericFilterBean`, which is a Spring-aware filter implementation. You are also using a Spring Security provided class (`AuthenticationTrustResolverImpl`) that allows you to make queries over the `Authentication` object to determine if it belongs to a certain level of authentication—for example, to determine if it is an anonymous user or a remember-me user.

Next let's define the custom filter in the `applicationContext-security.xml` file, and also create a new user that will have the role `ROLE_ADMIN`. So the `applicationContext-security` now looks like Listing 8-24.

Listing 8-24. applicationContext-security.xml with a Custom Filter

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/security
 http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true">
        <security:intercept-url pattern="/*" access="ROLE_USER" />
        <security:custom-filter ref="userAgentFilter"
            before="FILTER_SECURITY_INTERCEPTOR"/>
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user authorities="ROLE_USER"
                     name="car" password="scarvarez" />
                <security:user authorities="ROLE_ADMIN,ROLE_USER"
                     name="mon" password="scarvarez" />
            </security:user-service>
```

```
        </security:authentication-provider>
    </security:authentication-manager>
    <bean id="userAgentFilter" class="com.apress.pss.security.UserAgentFilter"/>
</beans>
```

As you can see from Listing 8-24, it is pretty easy to configure a custom filter in the security filter chain. You use the namespace element `<custom-filter>` and put in a reference to a filter implementation bean. Also, you can specify where the filter will be within the chain. In the example, you are specifying that it must execute before the `FilterSecurityInterceptor` filter, as at this point all the authentication filters have already run, which means you have a proper `SecurityContext` set up and authorization checking is about to begin.

Now if you restart the application—using Google Chrome, for example—and visit the URL `http://localhost:8080/hello` and log in with "car", and "scarvarez", you will receive an access-denied page in response to the message, "The browser you are using is not supported for your user role. Use Firefox instead." This is shown in Figure 8-8. If, on the other hand, you do the same with the Firefox browser, you will be allowed access to the now very familiar "Hello World" page. If you do the same but log in with "mon" and "scarvarez" and access the URL `http://localhost:8080/hello/`, you will be able to access the functionality of the page (the "Hello World" message) from any browser.



Figure 8-8. *An error you receive when logging in with Chrome instead of Firefox*

This was a simple example of configuring your own security filter to add additional behavior to the security implementation you are configuring in your application. You could think of many different implementations in which you could improve or modify the security treatment through the filter chain.

Handling Errors and Entry Points

Spring Security has a very nice error-handling mechanism built in. It offers a comprehensive set of exceptions that map to the most common cases of security errors you could expect to have in a system. Continuing with Spring Security's great single-responsibility architecture, the handling of error conditions is mostly encapsulated in one single class. This class is (and you have studied it before) the `org.springframework.security.web.access.ExceptionTranslationFilter`, and it basically deals with two types of exceptions: `AccessDeniedException` and `AuthenticationException`. If any other exception is caught by this filter, it will simply rethrow it as a `RuntimeException`.

Spring Security offers a couple of extension points you can use to plug in functionality in the form of a custom entry point and a custom access-denied handler. Basically, what happens is that when an `AuthenticationException` is caught by the filter, an `AuthenticationEntryPoint`'s "commence" method is called with the HTTP request, the HTTP response, and the exception. The particular `AuthenticationEntryPoint` implementation that is configured in the application can decide what to do with the exception and, more importantly, what to do with the HTTP response. By default, the implementation used is `org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint`, which redirects to the login URL, which as you already know is `/spring_security_login` in the root of your application.

The entry point is invoked when an `AuthenticationException` is thrown or when an `AccessDeniedException` is thrown and the current `Authentication` object is anonymous.

As another artificial example, you will implement an entry point that will add a cookie each time an authentication attempt is made from the client side and then show the Basic Authentication scheme in the browser. This means you will set a cookie in the response that will increment its value every time the entry point is invoked. Then the cookie is sent back to the server every time with the new value. This means that the counter for the attempts is not stored in the server at all, it is stored just in the client cookie and is sent back and forth between the client and the server. So the server receives the cookie, increments its value, and then sends it back. It is not really innovative in any way, but it allows me, again, to show that you can override the entry point to do different things.

Most likely, the entry point is overridden when a different kind of authentication scheme is being used. For example, one of the standard entry points applies when using authentication schemes that require to show a login page, while others (in particular the `BasicAuthenticationEntryPoint`) are used to set up particular values in the response to inform the browser how to treat it. In most cases, the implementation of an entry point goes hand in hand with the implementation of a new security filter. If you take a look at the implementations provided by the framework, you can see this clear relationship in `BasicAuthenticationEntryPoint` – `BasicAuthenticationFilter`, `LoginUrlAuthenticationEntryPoint` – `UsernamePasswordAuthenticationFilter` (kind of, this one is not as one to one), `CasAuthenticationEntryPoint` – `CasAuthenticationFilter`, and some others. Basically, the entry points set the groundwork and then the filter processes the subsequent request.

Figure 8-9 shows the relationship between the `ExceptionTranslationFilter` and the `AuthenticationEntryPoint`.

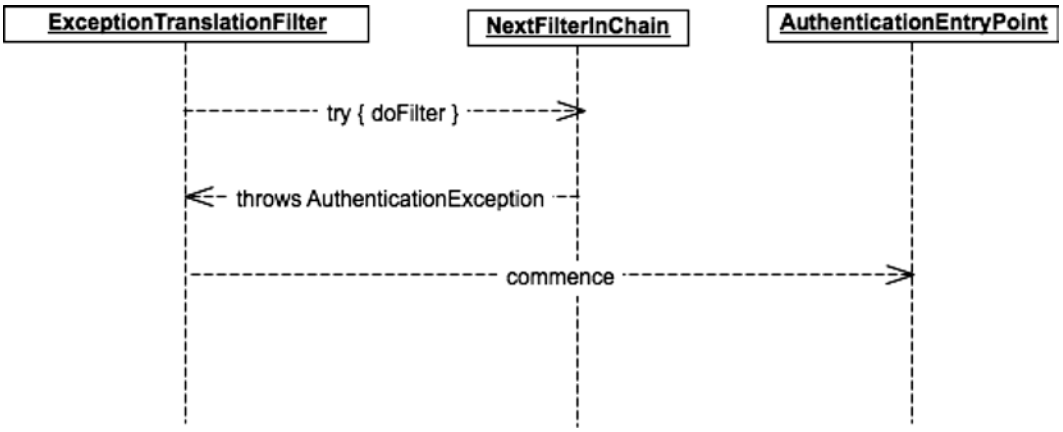


Figure 8-9. *The ExceptionTranslationFilter and AuthenticationEntryPoint relationship*

Listing 8-25 shows the simple entry-point implementation, and Listing 8-26 shows the configuration needed to make it work. It is a very simple implementation, but it is actually a little more complex than the standard `BasicAuthenticationEntryPoint`.

Listing 8-25. AuthenticationEntryPoint Implementation That Creates an Attempts Cookie and a Basic Authentication Response

```
package com.apress.pss.security;

import java.io.IOException;

import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
```

```
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;

public class AttemptsAuthenticationEntryPoint implements AuthenticationEntryPoint{

    public void commence(HttpServletRequest request,
        HttpServletResponse response, AuthenticationException authException)
        throws IOException, ServletException {
        response.addHeader("WWW-Authenticate", "Basic realm=\"theapp\"");
        response.addHeader("Set-Cookie",
          "authentication_attempts="+(getDeniesCookie(request)+1)+"; Max-Age=3600; Version=1");
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
           authException.getMessage());

    }

    private int getDeniesCookie(HttpServletRequest request) {
        for(Cookie cookie:request.getCookies()){
            if(cookie.getName().equals("authentication_attempts")){
                return Integer.parseInt(cookie.getValue());
            }
        }
        return 0;
    }

}
```

The code in Listing 8-25 has a lot in common with the code from `org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint` because it also tells the browser in the response to start a new basic authentication input process. The main difference is, of course, in the retrieval and processing of the `authentication_attempts` cookie. The cookie is first retrieved from the request in the private method `getDeniesCookie`. Then the value of this cookie (which is assumed to be an integer) is increased by one and reset in the response header in the following line:

```
response.addHeader("Set-Cookie",
  "authentication_attempts="+(getDeniesCookie(request)+1)+"; Max-Age=3600; Version=1");
```

Listing 8-26. Spring Configuration Needed to Use a Custom Entry Point

```
......
<security:http auto-config="true" entry-point-ref="attAuthenticationEntryPoint">
        <security:intercept-url pattern="/*" access="ROLE_USER" />
    </security:http>
...
<bean id="attAuthenticationEntryPoint"
class="com.apress.pss.security.AttemptsAuthenticationEntryPoint"/>
```

If a different kind of exception than the ones I mentioned before is thrown (particularly, an AccessDeniedException for a fully authenticated user), an AccessDeniedHandler is invoked instead of the AuthenticationEntryPoint.

The default AccessDeniedHandler implementation that is invoked is AccessDeniedHandlerImpl, which also by default will set a 403 error status in the response and let the browser render its default 403 page. However, you can also configure an errorPage property in the handler to determine that a forward (an internal dispatching mechanism inside the application, different from a redirect) is made to a customized error page, which is probably the most common personalization you will do when using the AccessDeniedHandler.

You can also define your own implementation of AccessDeniedHandler, and that is what you will do here to illustrate the point, but you will also use a custom errorPage property much as you would use it in the AccessDeniedHandlerImpl.

The implementation, as in the previous example, will simply add an extra header in the response in the form of a cookie that specifies the number of "access-denied"responses received from the particular computer and browser from where the requests are coming. Listing 8-27 shows the implementation of the handler. Most of the code is copied and pasted from the AccessDeniedHandlerImpl.

Listing 8-27. CookieAccessDeniedHandler That Sets a denied_counter Cookie on the Response and Then, Optionally, Uses an Error Page the Same Way as the AccessDeniedHandlerImpl Does

```
package com.apress.pss.security;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.Cookie;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.access.AccessDeniedException;
import org.springframework.security.web.WebAttributes;
import org.springframework.security.web.access.AccessDeniedHandler;

public class CookieAccessDeniedHandler implements AccessDeniedHandler {

    private static final String ACCESS_DENIES = "access_denies";

    private String errorPage;

    public void handle(HttpServletRequest request,
            HttpServletResponse response,
            AccessDeniedException accessDeniedException) throws IOException,
            ServletException {
        if (!response.isCommitted()) {
            response.addCookie(new Cookie(ACCESS_DENIES,
                 String.valueOf(getDeniesCookie(request)+1)));
            if (errorPage != null) {
                request.setAttribute(WebAttributes.ACCESS_DENIED_403, accessDeniedException);
                response.setStatus(HttpServletResponse.SC_FORBIDDEN);
                RequestDispatcher dispatcher = request.getRequestDispatcher(errorPage);
                dispatcher.forward(request, response);
            } else {
```

```
                response.sendError(HttpServletResponse.SC_FORBIDDEN,
accessDeniedException.getMessage());
            }
        }

    }

    private int getDeniesCookie(HttpServletRequest request) {
        for(Cookie cookie:request.getCookies()){
            if(cookie.getName().equals(ACCESS_DENIES)){
                return Integer.parseInt(cookie.getValue());
            }
        }
        return 0;
    }

}
```

As I said, the code is mostly the same as the `AccessDeniedHandlerImpl`. I just added the cookie in the response, which will be set to incremental values every time this handler is invoked. You can see that there is also the logic for processing the `errorPage` property in case it is set. A servlet dispatcher forward will be done to this error page URL, which means that it will have access to the same request that is used inside this class.

Changing the Security Interceptor

The security interceptor is a class you rarely find yourself modifying or replacing, as the default implementations cover the most common scenarios of filter security and method-level security. However, you can extend it for use in different kinds of applications that don't strictly fit into the web app-business method services scheme. For example, Spring Integration has its own security interceptor implementation in the form of the `org.springframework.integration.security.channel.ChannelSecurityInterceptor`. Instead of working with simple method invocations or filter invocations, it works with a different abstraction, which is the `org.springframework.integration.security.channel.ChannelInvocation`. This means that it basically intercepts send and receive calls on a determined secured channel. Spring Integration also uses the class `org.springframework.integration.security.channel.ChannelSecurityMetadataSource` as the `org.springframework.security.access.SecurityMetadataSource` implementation for message channels.

The creation of a different security interceptor, as the Spring Integration example shows, is basically for when you want to add Spring Security's authorization support to applications that don't follow the standard web-service way of doing things. However, keep in mind that a previous authentication mechanism must be in place, as the security interceptor will look for the different components that it needs to grant access to a resource. This means that an `Authentication` object must exist in the `SecurityContext`, the `AccessDecisionManager` must be configured, and so on. Listing 8-28 shows the security interceptor implementation from Spring Integration.

Listing 8-28. ChannelSecurityInterceptor Is a Spring Integration Custom Implementation of the Security Interceptor, Which Knows About Intercepting ChannelInvocation

```
/*
 * Copyright 2002-2010 the original author or authors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
```

```
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package org.springframework.integration.security.channel;

import java.lang.reflect.Method;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.security.access.SecurityMetadataSource;
import org.springframework.security.access.intercept.AbstractSecurityInterceptor;
import org.springframework.security.access.intercept.InterceptorStatusToken;
import org.springframework.util.Assert;

/**
 * An AOP interceptor that enforces authorization for MessageChannel send and/or receive calls.
 *
 * @author Mark Fisher
 * @author Oleg Zhurakousky
 */
public class ChannelSecurityInterceptor extends AbstractSecurityInterceptor implements
MethodInterceptor {

    private final ChannelSecurityMetadataSource securityMetadataSource;


    public ChannelSecurityInterceptor(ChannelSecurityMetadataSource securityMetadataSource) {
        Assert.notNull(securityMetadataSource, "securityMetadataSource must not be null");
        this.securityMetadataSource = securityMetadataSource;
    }


    @Override
    public Class<?> getSecureObjectClass() {
        return ChannelInvocation.class;
    }


    public Object invoke(MethodInvocation invocation) throws Throwable {
        Method method = invocation.getMethod();
        if (method.getName().equals("send") || method.getName().equals("receive")) {
            return this.invokeWithAuthorizationCheck(invocation);
        }
        return invocation.proceed();
    }
```

Download from Wow! eBook <www.wowebook.com>

```
    private Object invokeWithAuthorizationCheck(MethodInvocation methodInvocation) throws
Throwable {
        Object returnValue = null;
        InterceptorStatusToken token = super.
          beforeInvocation(new ChannelInvocation(methodInvocation));
        try {
            returnValue = methodInvocation.proceed();
        }
        finally {
            returnValue = super.afterInvocation(token, returnValue);
        }
        return returnValue;
    }


    @Override
    public SecurityMetadataSource obtainSecurityMetadataSource() {
        return this.securityMetadataSource;
    }

}
```

In the code, you can see that `ChannelInvocation` wraps a `MethodInvocation` before calling the `beforeInvocation` method on the parent class. Apart from wrapping, it will also do some inner processing to make the current executing channel available to be queried for `ConfigAttributes`. Also note that the interceptor will do its security validations only if the method being invoked is one of the standard "send" and "receive" methods of Spring Integration.

The `ChannelSecurityMetadataSource` that is being used in the code is the one that will be queried for obtaining the security metadata attributes. It is the one that knows how to extract this information from Spring Integration Channels and the related `org.springframework.integration.security.channel.ChannelAccessPolicy`.

It is very unlikely you will override the security interceptor in your own applications. However, it is good to know that you could do it and also understand why you would want to do it.

As the example for Spring Integration shows, one reason you would want to replace a security interceptor (or add an additional one) is because you have certain abstractions, to which you want to apply interception-based security, that don't fit either URL interception or simple method interceptions. You want to give a more meaningful name to your interception logic and also filter certain things that are not filtered by default with the default implementations. That is what the Spring Integration example is doing. It is intercepting `MessageChannel` communication, which is the domain element that makes sense in its context; However, in the end, it is basically intercepting methods and filtering to intercept only the methods `send` and `receive`, which again are the ones that make sense in the particular context.

Spring Security Extensions Project

There is a whole project dedicated to the development of Spring Security extensions, where people from the community can develop their own extensions on top of Spring Security. In this way, they can decouple these extensions from the main Spring Security project, allowing it to evolve independently. The home of this extensions project is http://static.springsource.org/spring-security/site/extensions.html.

If you visit that page, you will see that currently there are two extensions projects in existence: Kerberos integration and SAML2 integration. There is also the OAuth integration, which lives on its own as an individual project. You can find it at http://static.springsource.org/spring-security/oauth/.

Each project in Spring Security Extensions can implement as many parts of the framework as it needs in order to work correctly. This means that many of the extension points that I defined in this chapter (and some others) could be overridden in a particular extension in order to do its work. For example, OAuth uses a custom filter `OAuth2AuthenticationProcessingFilter` and a custom user details service (actually, a `ClientDetailsService`), among others. SAML creates its own `AuthenticationProvider` in the form of `SAMLAuthenticationProvider` and also has its entry point as `SAMLEntryPoint`, among other implementations.

Summary

In this chapter, I showed you how the modularity in the architecture of Spring Security pays off when you want to customize or extend its behavior. I showed some of the different and most common extension points that Spring Security offers so that you can adapt its functionality to your particular application while keeping the core functionality, making the work easier for you by leveraging this functionality. After reading this chapter, and with all the theory and practice from previous chapters, you should feel confident enough to implement functionality that goes beyond the out-of-the-box offerings of the framework.

Note that this chapter does not include a comprehensive list of extension points in the framework. Remember that you can get the source code of Spring Security, which means that you can change absolutely everything to adapt it to your own needs. You won't often need to change any core aspects of the framework, but it is good to know that you could. Keep in mind that sometimes working with Spring Security is like putting together a puzzle. In each case, you are assembling components, but in the case of Spring Security, you can replace some of the default components with customized ones to change its behavior.

CHAPTER 9

Integrating Spring Security with Other Frameworks and Languages

This chapter will explore Spring Security in the context of other application frameworks and languages that run on the JVM.

You saw in previous chapters that the two main ways of using Spring Security are in the web layer in the shape of filters and in the business layer with Spring AOP. This means that you could use Spring Security in any application that is built on top of the Servlet technology or in any application that is willing to use Spring and Spring AOP to handle its object life cycles and interactions.

In the following sections, you'll see examples of both cases. We'll start by looking at a couple of popular Java frameworks (one which is also Spring based) and how to use Spring Security with them. These frameworks are the popular Struts 2 web framework and Spring Web Flow, another member of the SpringSource suite.

After studying these two frameworks, we'll take a brief look at a few Java Virtual Machine (JVM) programming languages (and some of their related frameworks) and how to use Spring Security with them. We'll be looking at Groovy in the context of its web-development framework Grails, JRuby in the context of Rails, and Scala embedded in a Spring web application.

I will not go into any of these frameworks or languages in much detail, because that would be beyond the scope of this book. The purpose of this chapter is simply to explain how to use Spring Security in a wider context.

Spring Security with Struts 2

I decided to talk about Struts 2 here because, since I started working in IT, Struts and Struts 2 have been the frameworks (apart from Spring itself, of course) I have come across most often. True, Struts is not as common as it used to be, with new and more friendly frameworks coming out all the time. Even version 2 is getting rather old, although it is still well maintained. (The version used on this book is from November 19, 2012.) However, it is still widely used and is as good a candidate as any other external Servlet-based framework to integrate with Spring and Spring Security.

Struts 2 is a popular Java web framework built by merging the original Struts project and the WebWork project. Struts 2 was always intended to be a complete evolution from the original Struts framework, adapted to a new generation of powerful web frameworks, and it really kept little of the original Struts principles and implementations.

Struts 2 is an MVC (model view controller) framework built on top of the standard Java Servlets technology, so many of the web-based security principles you have read about in this book apply without modification. If you want to use only the URL-level, web-based security you studied in Chapter 4, you just need to configure your web.xml and your Spring Security filter chain accordingly.

As with the rest of frameworks and languages in this chapter, I won't explain Struts 2 in any depth (because there are many good books available on that subject). I'll explain just enough so that you can use Spring Security with it. In fact, I assume that if you are reading this section it is probably because you are already using Struts 2 and want to integrate Spring Security with it. However, to explain a little more, Figure 9-1 shows the big picture of how Struts 2 works.

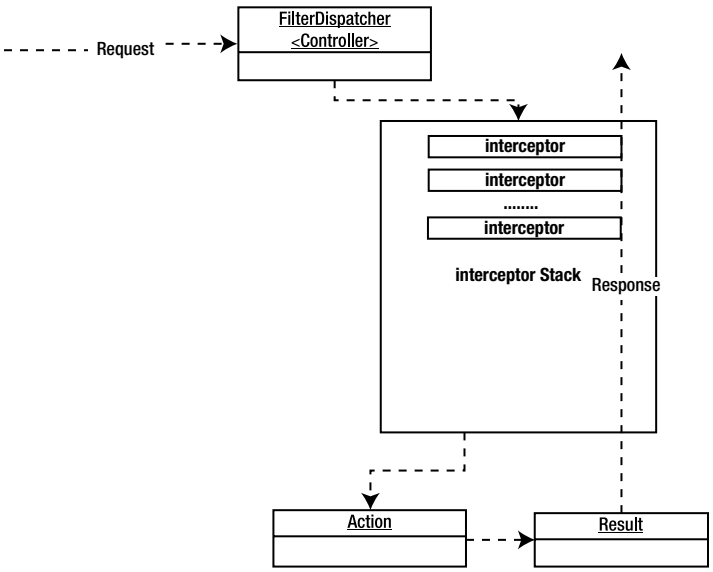


Figure 9-1. *How Struts 2 works*

When a request comes to the application, a front controller (http://en.wikipedia.org/wiki/Front_Controller_pattern) implemented in a Servlet filter takes care of the request. It sends the request through a set of configured interceptors that perform different kinds of functionality before and after the action is invoked. The action is then invoked, which carries with it all the business logic required by the current request. After the action finishes, a result object representing the view is created and the interceptor stack is invoked in inverse order as before (while returning the result) until a response is finally returned to the client.

We'll be using Maven to build our project. The first thing we'll do is create a new project. From the command line in a directory of your choice, execute the command `mvn archetype:generate -DarchetypeCatalog=http://struts.apache.org/2.3.7/`. Then follow the prompts by selecting a new *starter project* at the first prompt and selecting some relevant `groupId` and `artifactId` names. (I chose `com.apress.pss.struts` and `struts-example`, respectively. You could use the same to follow along in the examples easily).

Executing the last Maven command is a fantastic first step for creating our example because it creates a simple, functional Struts 2 web application with the components we need to create our test, including Spring integration (although it uses an old DTD version of the configuration XML file). The file structure we have after executing that command is shown in Figure 9-2.

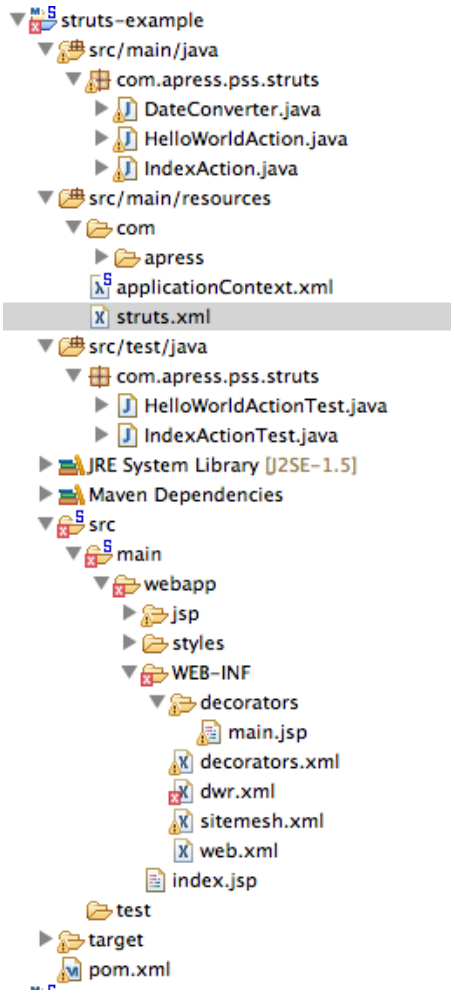


Figure 9-2. *Struts 2 application file structure after executing the Maven command*

In the preceding figure, you can see that there is already an `applicationContext.xml` file (the Spring configuration file) created for you inside the resources directory. You can also see inside that directory, a file named `struts.xml`; this is the main Struts 2 configuration file, and you can see its contents in Listing 9-1.

Listing 9-1. The struts.xml file

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration 2.3//EN"
    "http://struts.apache.org/dtds/struts-2.3.dtd">

<struts>

    <constant name="struts.enable.DynamicMethodInvocation" value="false"/>
```

```
    <package name="myPackage" extends="struts-default">
        <action name="index" class= "com.apress.pss.struts.IndexAction">
            <result>/jsp/index.jsp</result>
        </action>
        <action name="helloWorld" class="helloWorldAction">
            <result name="input">/jsp/index.jsp</result>
            <result>/jsp/helloWorld.jsp</result>
        </action>
    </package>
</struts>
```

In Listing 9-1, the important part to note is the definition of the two actions. In the first action, "index" you can see that in the `class` attribute of the `action` element, a class name is defined. That class is defined in `com.apress.pss.struts.IndexAction`.

The second action, on the other hand, doesn't define a class name on the attribute `class`. In this case, it defines a simple string that identifies a Spring bean in the `applicationContext.xml` that you'll see later. When a request comes in, the framework looks at the definitions from this file to determine how to handle the particular request. (This definition is simplistic, but it's good enough to make the point.) If it sees the name of a class in the `action` attribute, the framework instantiates a new object of that class to handle the request. If the framework detects that it is not a class, it looks for a bean with that ID in the Spring configuration and gets the object from there. All this is possible in the framework thanks to the `struts2-spring-plugin` plugin, which enhances Struts 2 with Spring functionality and makes it possible to define actions (and other components such as interceptors) in Spring beans.

What we'll do now is rewrite the default `applicationContext.xml`, replace it with the one from Listing 9-2, and add the familiar dependencies from Listing 9-3 to the `pom.xml` file.

Listing 9-2. The applicationContext.xml file for the Struts 2 application

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

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

http://www.springframework.org/schema/security
http://www.springframework.org/schema/security/spring-security-3.1.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

    <bean id="helloWorldAction" class="com.apress.pss.struts.HelloWorldAction" scope="prototype" />

    <security:http auto-config="true" />

    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
```

```
                <security:user name="car" authorities="ROLE_USER"
                    password="scarvarez" />
                <security:user name="mon" authorities="ROLE_ADMIN"
                    password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans>
```

Listing 9-3. Spring Security dependencies in the pom.xml file

```
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.1.3.RELEASE</version>
</dependency>
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.1.1</version>
</dependency>
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.3</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>3.0.6.RELEASE</version>
</dependency>
```

Next, you need to add the Spring Security filter to the `web.xml`. This filter should execute before the Struts 2 filter, so an authentication object is populated before reaching the action. Remember that Struts 2 works with filters and not with servlets. Listing 9-4 shows the `web.xml` file you can use as a test. I also removed some filter definitions I don't care to use in the example.

Listing 9-4. The web.xml with the Spring Security filter and Struts 2 filter

```
<?xml version="1.0" encoding="UTF-8"?>

<web-app id="starter" version="2.4"
         xmlns="http://java.sun.com/xml/ns/j2ee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
         http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Struts 2 - Maven Archetype - Starter</display-name>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath*:applicationContext*.xml</param-value>
    </context-param>

    <!-- Filters -->

     <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter>
        <filter-name>action2</filter-name>
        <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter
        </filter-class>
    </filter>
     <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <filter-mapping>
        <filter-name>action2</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>

    <!-- Listeners -->
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>

    <!-- Servlets -->
    <servlet>
        <servlet-name>dwr</servlet-name>
        <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
        <init-param>
            <param-name>debug</param-name>
```

```
            <param-value>true</param-value>
        </init-param>
    </servlet>
    <servlet>
        <servlet-name>jspSupportServlet</servlet-name>
        <servlet-class>org.apache.struts2.views.JspSupportServlet</servlet-class>
        <load-on-startup>5</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dwr</servlet-name>
        <url-pattern>/dwr/*</url-pattern>
    </servlet-mapping>

    <!-- Welcome file lists -->
    <welcome-file-list>
        <welcome-file>index.jsp</welcome-file>
        <welcome-file>default.jsp</welcome-file>
        <welcome-file>index.html</welcome-file>
    </welcome-file-list>

</web-app>
```

Next, I simplified the jsp files inside the jsp folder like this: the file helloWorld.jsp just contains the string "Pagina Segura" inside, nothing else.

Struts 2 has a built-in system to handle the exceptions that might be thrown in your application. This mechanism would get in the way of Spring Security's exception-handling system, which depends on `AccessDeniedException`, among other exceptions, to be thrown to alter the flow of execution. For example, when showing a login form, you need to deactivate the Struts 2 exception-handling mechanism. To do that in the file `struts.xml`, you add the line `<constant name="struts.handle.exception" value="false" />` just below the other line that contains a defined `<constant>` element.

Finally, we need to secure our action. Make your `HelloWorldAction` look like Listing 9-5.

Listing 9-5. HelloWorldAction secured

```
package com.apress.pss.struts;

import java.util.Date;

import org.springframework.security.access.annotation.Secured;

import com.opensymphony.xwork2.ActionSupport;
import com.opensymphony.xwork2.validator.annotations.Validation;
import com.opensymphony.xwork2.validator.annotations.RequiredStringValidator;
import com.opensymphony.xwork2.validator.annotations.RequiredFieldValidator;
import com.opensymphony.xwork2.conversion.annotations.Conversion;
import com.opensymphony.xwork2.conversion.annotations.TypeConversion;
```

```
public class HelloWorldAction extends ActionSupport {

    @Secured("ROLE_USER")
    public String execute() throws Exception {
        return SUCCESS;
    }
}
```

Now if you restart your application and visit `http://localhost:8080/struts-example/helloWorld`, you'll be presented with the standard Spring Security login form. If you use the login username **car** and password **scarvarez**, you'll be able to access the page shown in Figure 9-3.

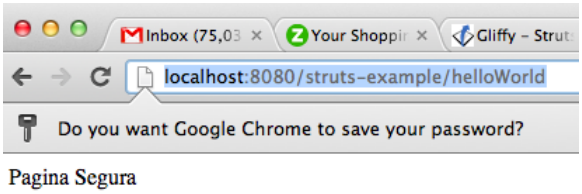


Figure 9-3. *The secured Struts 2 application and the accessed action*

Download from Wow! eBook <www.wowebook.com>

Spring Security with Spring Web Flow

Spring Web Flow is a framework, built on top of Spring MVC, that allows you to link different steps of a web-driven process into a fluent workflow. In other words, it allows you to define in a declarative way the different steps that a web application can go through while you are interacting with it. Basically, you use it to define a set of rules and transitions between the user interface (UI) parts of a web application and the back-end process that each transition should trigger.

Graphically, Spring Web Flow works, in a simplified form, as shown in Figure 9-4. The example is a fake web page for a simplified product. The boxes represent various states (the View state, Action state, Decision state, Subflow state, and others), and the arrows represent transitions.

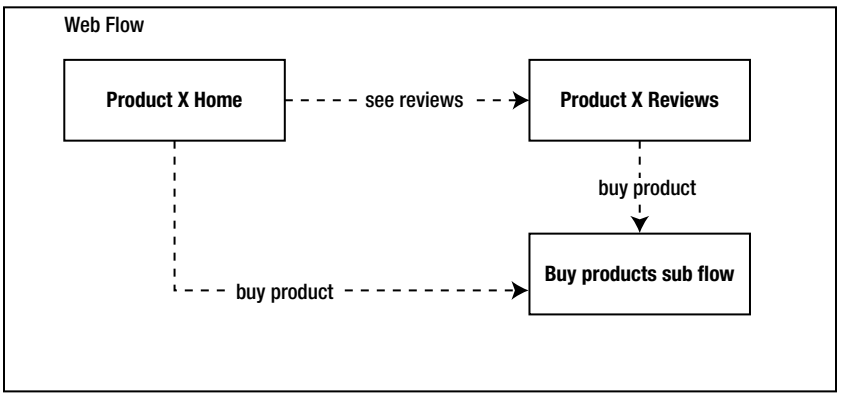


Figure 9-4. *Simple Spring Web Flow scheme showing that, from a product page, you can go to the review page or buy the product*

To implement this simple flow with Spring Web Flow, we'll create a new project. As is the case for most of the examples, we'll use Maven to build and manage our project. So let's do it. From the command line in a place where you want to create your project, execute the following:

```
mvn archetype:create -DarchetypeGroupId=org.apache.maven.archetypes\
-DarchetypeArtifactId=maven-archetype-webapp -DarchetypeVersion=1.0\
-DgroupId=com.apress.pss -DartifactId=webflow-example -Dversion=1.0-SNAPSHOT
```

That will create a new project named `webflow-example` in the directory.

Now we'll create all the configuration files. Replace the `pom.xml` file in the new project with the one shown in Listing 9-6, and replace the `web.xml` file with the one shown in Listing 9-7. Then, in the `WEB-INF` directory, create the files shown in Listings 9-8 and 9-9 with the names `products-servlet.xml` and `applicationContext-security.xml`, respectively.

Listing 9-6. The pom.xml file with Spring Security and Spring Web Flow dependencies

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.pss</groupId>
    <artifactId>webflow-example</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>webflow-example Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <properties>
        <org.springframework.version>3.1.1.RELEASE</org.springframework.version>
    </properties>
    <dependencies>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-core</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-beans</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-aop</artifactId>
            <version>${org.springframework.version}</version>
        </dependency>
```

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${org.springframework.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context-support</artifactId>
    <version>${org.springframework.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${org.springframework.version}</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>${org.springframework.version}</version>
</dependency>
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.0.1</version>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.5</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-core</artifactId>
    <version>3.1.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.1.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.1.0.RELEASE</version>
</dependency>
```

```
        <dependency>
            <groupId>org.springframework.webflow</groupId>
            <artifactId>spring-webflow</artifactId>
            <version>2.3.1.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.3</version>
        </dependency>

    </dependencies>
    <build>
        <plugins>
            <plugin>
                <groupId>org.mortbay.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>8.1.1.v20120215</version>
                <configuration>
                    <connectors>
                        <connector implementation="org.eclipse.jetty.server.nio.
                        SelectChannelConnector">
                            <port>8080</port>
                            <maxIdleTime>60000</maxIdleTime>
                        </connector>
                    </connectors>
                </configuration>
            </plugin>
        </plugins>
        <finalName>webflow-example</finalName>
    </build>
</project>
```

Listing 9-7. The web.xml for Spring Web Flow and Spring Security

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

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

```
<listener>
 <listener-class>
            org.springframework.web.context.ContextLoaderListener
 </listener-class>
</listener>

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

<filter-mapping>
 <filter-name>springSecurityFilterChain</filter-name>
 <url-pattern>/</url-pattern>
</filter-mapping>

<servlet>
 <servlet-name>products</servlet-name>
 <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
</servlet>

 <servlet-mapping>
  <servlet-name>products</servlet-name>
  <url-pattern>/</url-pattern>
 </servlet-mapping>
</web-app>
```

You should already be familiar with the content of Listing 9-7 It is a `web.xml` file that includes Spring's `ContextLoaderListener`, which loads the Spring application-context file given in `context-param` "`contextConfigLocation`". It also defines Spring's `DispatcherServlet` servlet, which takes care of setting up Spring MVC by loading the appropriate configuration file. We are also defining the already familiar `springSecurityFilterChain` filter. Both the security filter and the dispatcher servlet are configured to handle every URL in the system.

■ **Note** In Spring MVC, the name of the DispatcherServlet servlet is important because that name will match the name of the Spring configuration file that will be used in the application to configure the application. For example, in our case, by defining the DispatcherServlet with name "products", Spring will expect to find a file with the name *products-servlet.xml* in the WEB-INF folder where the beans for the web layer should be defined.

Listing 9-8. The products-servlet.xml file that imports the flow

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
<import resource="example-webflow.xml" />
</beans>
```

This file is very simple, and its only job is to import another file (the `example-webflow.xml` file), which will contain the entire Spring Web Flow configuration. This configuration will remain in a different file just to keep it separated from the main servlet file.

Listing 9-9. The applicationContext-security.xml file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true"/>
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user authorities="ROLE_USER" name="car" password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans>
```

This is another file you should be able to understand easily by now. Listing 9-9 shows a very basic Spring Security configuration. We are defining just a single user with role `ROLE_USER`, which will be enough for our tests. We are not defining any URL security rules here because that is not what we want to do in this Spring Web Flow example. We want to add security at the flow level (its states), and that is what I will show you how to do.

Now we need to define the web-flow configuration of the application as well as the actual web flows themselves. Again, this will be a simplistic example just to show how the functionality works. The example will be based on Figure 9-4. Listing 9-10 shows the web-flow configuration file, and Listing 9-11 shows our only flow definition.

Listing 9-10. The example-webflow.xml in the WEB-INF folder

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:webflow=
    "http://www.springframework.org/schema/webflow-config"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/webflow-config
        http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd">

    <webflow:flow-builder-services id="flowBuilderServices"
         view-factory-creator="flowResourceFlowViewResolver" />

    <bean id="flowResourceFlowViewResolver"
        class="org.springframework.webflow.mvc.builder.MvcViewFactoryCreator">
    </bean>

    <webflow:flow-executor id="flowExecutor" flow-registry="flowRegistry">
    </webflow:flow-executor>
```

```
    <webflow:flow-registry flow-builder-services="flowBuilderServices"
        id="flowRegistry">
        <webflow:flow-location path="/WEB-INF/flows/products/product.xml" />
    </webflow:flow-registry>

    <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerAdapter">
        <property name="flowExecutor" ref="flowExecutor" />
    </bean>

    <bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
        <property name="flowRegistry" ref="flowRegistry" />
        <property name="order" value="0" />
    </bean>

</beans>
```

This file defines the general configuration for Spring Web Flow. The main part of the file is the element `flow-registry and its attribute flow-builder-services`. This element is where the location of the flows in the application are defined. Currently, we'll define only one flow which will be in the product.xml file. Also, note the way our views will get resolved when referenced in a view state in a flow. The class `org.springframework.webflow.mvc.builder.MvcViewFactoryCreator` is the one that will resolve view locations. By default, it will resolve view files by looking in the flow definition directory for files whose names are the names of the view states concatenated with *.jsp* at the end. This simply means that if a view is named *review*, it will look for a file named *review.jsp*.

Listing 9-11. The product.xml simple flow in the /WEB-INF/flows/products folder

```
<flow xmlns="http://www.springframework.org/schema/webflow"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/webflow
        http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

 <view-state id="main">
   <transition on="review" to="review" />
   <transition on="buy" to="buy" />
 </view-state>

 <view-state id="review">
   <transition on="buy" to="buy" />
 </view-state>

  <view-state id="buy" />
</flow>
```

This is the actual flow definition. We are simply defining three view states here. As I said before, each of these view states will map to a physical view file in the application. The view files, by default behavior, should be located in the same directory as this flow file and should be named according to the view states concatenated with *.jsp* at the end. So we should have, in the `WEB-INF/flows/products` directory files named `main.jsp`, `review.jsp`, and `buy.jsp`. They are shown in Listings 9-12, 9-13, and 9-14, respectively.

Listing 9-12. The main.jsp for the main view state

```
<head>
  <title>Ferrari f40</title>
</head>
<body>
  <h2>F40</h2>
  <img src="http://www.instablogsimages.com/1/2012/09/14/1347613392.jpg"/>
  <a href="${flowExecutionUrl}&_eventId=review">Review</a>
  <a href="${flowExecutionUrl}&_eventId=buy">Buy</a>
</body>
</html>
```

Listing 9-13. The review.jsp for the review state

```
<head>
  <title>Ferrari f40</title>
</head>
<body>
  <h2>Ferrari F40 runs really fast</h2>
  <a href="${flowExecutionUrl}&_eventId=buy">Buy</a>
</body>
</html>
```

Listing 9-14. The buy.jsp for the buy view state of the flow

```
<head>
  <title>Ferrari f40</title>
</head>
<body>
  <h2>You just bought the Ferrari F40. We will deliver gift wrapped tomorrow between 9am and 11am</h2>
</body>
</html>
```

In the previous three JSP files, we defined the three view states in our web flow. Listings 9-12 and 9-13 have transition triggers in the form of the `eventId` parameter in the `href` links. `${flowExecutionUrl}` makes reference to the current executing state of the flow. So, in the case of the main state the first time you access the page, the value of `${ flowExecutionUrl }` is `/product?execution=e1s1`, making the full URL to the next flow states something like `/product?execution=e1s1}&_eventId=review`.

If you visit `http://localhost:8080/product`, you'll be taken to the main view state. That view displays `main.jsp`, where you'll see the Ferrari picture and two links as Figure 9-5 shows. At the moment, you can click on both links—`Review` and `Buy`—without a problem. In a real application, you probably would want to allow anybody to read the product review but limit the ability to buy the product only to authenticated users.

F40



Review Buy

Figure 9-5. *The main.jsp, which is the entry point into the web flow*

Let's use Spring Security to do just that. Spring Security and Spring Web Flow integrate nicely because both are part of the Spring portfolio.

Securing flows with Spring Security is easy. First we need to add to the current `<webflow:flow-executor>` element from the file `example-webflow.xml` (the element in Listing 9-15) and add the content shown in Listing 9-16 somewhere in the same file.

Listing 9-15. The flow-executor with Spring Security Listener

```
<webflow:flow-execution-listeners>
     <webflow:listener ref="securityFlowExecutionListener" />
</webflow:flow-execution-listeners>
```

Listing 9-16. The Spring Security Listener bean

```
<bean id="securityFlowExecutionListener"
     class="org.springframework.webflow.security.SecurityFlowExecutionListener" />
```

By implementing the preceding content, you integrate Spring Security into Spring Web Flow. All you need to do now is decide which parts of the flow to secure and what constraints to define to secure them. But before that, I'll explain briefly what this `SecurityFlowExecutionListener` is doing for Spring Webflow.

Spring Web Flow offers an abstraction in the form of the interface `org.springframework.webflow.execution.FlowExecutionListener`, which allows you to implement classes to listen to or observe the life cycle of a flow execution. When a listener implementation is registered for the flows, it can intercept the flow execution at different points in its life cycle. In that sense, it is similar to the AOP concepts you studied before.

`org.springframework.webflow.security.SecurityFlowExecutionListener` is a listener implementation that intercepts three particular points in the life cycle of the flow: `"sessionCreating"`, `"stateEntering"`, and `"transitionExecuting"`. In each of these interceptions, the listener delegates to a configured `org.springframework.security.access.AccessDecisionManager` like the ones I explained in different parts of this book. If an `AccessDecisionManager` implementation is not provided when the bean is defined, a new role-based access decision manager will be created on the fly.

So now we want to secure the view-state `"buy"` and allow access to it only to authenticated users with role `ROLE_USER`. To do that, you simply define the element `<secured attributes="ROLE_USER" />` as a child of the `"buy"<view-state>` element. After you do that, if you restart the application and try to access the `"buy"` state, you'll find yourself presented with the familiar Spring Security login screen. If you log in with the username **car** and the password **scarvarez**, you'll be able to reach the `"buy"` state. The resulting `"buy"` state view is shown in Figure 9-6.

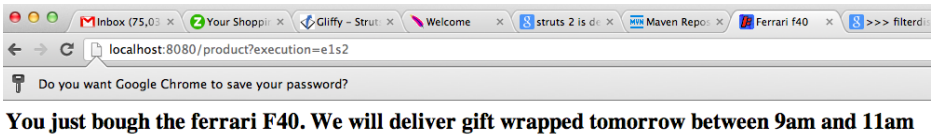


Figure 9-6. *The Ferrari has been bought in the buy state*

You could also secure the whole flow if you want by having the `<secured>` element be a direct child of the `<flow>` element. This means that, to access any state of the flow, a user needs to have the required permissions. Go ahead and try it yourself—it should be very straightforward.

SpEL-Based Security with Spring Web Flow

As I said before, Spring Web Flow configures an `AccessDecisionManager` for you if you don't explicitly inject one in the `SecurityFlowExecutionListener`. This default access decision manager uses a `RoleVoter`. This means that, by default, you can't use SpEL expressions in your configuration. Actually, Spring Web Flow integration with Spring Security currently supports only this role-based security. However, defining your own support for SpEL expression–based security is straightforward. To do this, you need to define a new `AccessDecisionVoter`, a new `SecurityExpressionHandler`, and a new `SecurityExpressionRoot`.

Let's configure a custom `AccessDecisionManager` so that you can use SpEL expressions. To configure it, simply add the bean definition from Listing 9-17 somewhere in the `example-webflow.xml` file. Then add the element `<property name="accessDecisionManager" ref="accessDecisionManagerForFlow"/>` as a child of the bean with the ID `"securityFlowExecutionListener"`. Next, change the `<secured>` element you defined before with the `<secured attributes="hasRole('ROLE_USER')" />`. Note how we are using a SpEL expression this time.

Now you need to create the new classes to support expressions at the flow state level. Currently, handlers exist for web-level expressions and method invocation-level expressions, which you studied previously in Chapters 4 and 5. In Spring Web Flow, there is a new abstraction—the state (a view state in our case). So you need to provide the support for Spring Security to be able to handle this.

Listing 9-18 shows the new `AccessDecisionVoter`. Listing 9-19 shows the new `SecurityExpressionHandler`, and Listing 9-20 shows the new `SecurityExpressionRoot`.

Listing 9-17. Bean definition for an Access Decision Manager that supports SpEL

```
<bean id="accessDecisionManagerForFlow"
    class="org.springframework.security.access.vote.AffirmativeBased">
    <property name="decisionVoters">
        <list>
            <bean
                class="com.apress.pss.webflow.security.StateExpressionVoter"/>
        </list>
    </property>
</bean>
```

Listing 9-18. StateExpressionVoter, which creates a SpEL evaluation context

```
package com.apress.pss.webflow.security;

import java.util.Collection;

import org.springframework.expression.EvaluationContext;
import org.springframework.expression.Expression;
```

```
import org.springframework.expression.ExpressionParser;
import org.springframework.expression.spel.standard.SpelExpressionParser;
import org.springframework.security.access.AccessDecisionVoter;
import org.springframework.security.access.ConfigAttribute;
import org.springframework.security.access.expression.ExpressionUtils;
import org.springframework.security.access.expression.SecurityExpressionHandler;
import org.springframework.security.core.Authentication;
import org.springframework.webflow.engine.State;

public class StateExpressionVoter implements AccessDecisionVoter<State> {

    private SecurityExpressionHandler<State> expressionHandler =
          new DefaultFlowStateSecurityExpressionHandler();

    public boolean supports(ConfigAttribute attribute) {
        return true;
    }

    public boolean supports(Class<?> clazz) {
        return true;
    }

    public int vote(Authentication authentication, State object,
            Collection<ConfigAttribute> attributes) {
         EvaluationContext ctx = expressionHandler.
                createEvaluationContext(authentication, object);
         ExpressionParser parser = new SpelExpressionParser();
         Expression exp = parser.parseExpression(attributes.iterator().next().getAttribute());
         return ExpressionUtils.evaluateAsBoolean(exp, ctx) ?
                    ACCESS_GRANTED : ACCESS_DENIED;
    }

}
```

Download from Wow! eBook <www.wowebook.com>

Listing 9-19. DefaultFlowStateSecurityExpressionHandler—the security expression handler

```
package com.apress.pss.webflow.security;

import org.springframework.security.access.expression.AbstractSecurityExpressionHandler;
import org.springframework.security.access.expression.SecurityExpressionRoot;
import org.springframework.security.core.Authentication;
import org.springframework.webflow.engine.State;

public class DefaultFlowStateSecurityExpressionHandler extends AbstractSecurityExpressionHandler
<State>{

    @Override
    protected SecurityExpressionRoot createSecurityExpressionRoot(
            Authentication authentication, State invocation) {
        SecurityExpressionRoot root = new StateSecurityExpressionRoot(authentication, invocation);
```

```
        root.setPermissionEvaluator(getPermissionEvaluator());
        return root;
    }

}
```

Listing 9-20. StateSecurityExpressionRoot, which extends the standard one

```
package com.apress.pss.webflow.security;

import org.springframework.security.access.expression.SecurityExpressionRoot;
import org.springframework.security.core.Authentication;
import org.springframework.webflow.engine.State;

public class StateSecurityExpressionRoot extends SecurityExpressionRoot {
    private State state;
    public StateSecurityExpressionRoot(Authentication a, State state) {
        super(a);
        this.state = state;
    }

}
```

Listings 9-19 and 9-20 show how to configure a new expression-evaluation context in Spring Security. As you saw in Chapter 8, Spring Security offers many extension points, both intentional and unintentional. In this particular case, I am taking advantage of the knowledge of the internal application programming interfaces (APIs) to develop support for SpEL for Spring Web Flow. You can see in the defined classes that I explicitly typed them with the `org.springframework.webflow.engine.State` class in the generics. This is how I am making the framework aware of this class for evaluating SpEL against it.

The support I just created in this example is very simplified and might not be production ready, but it does a good job of showing the functionality. Refer to Chapters 4 and 5 for more detailed explanations of SpEL.

If you now restart the application, you should get the same behavior as before. The `"buy"` state will be secured, and you'll need to log in with a user with the role `ROLE_USER` to access it.

Spring Security in Other JVM Languages

I am not an expert in any of the languages that follow, and maybe I won't be using them in the examples in the most idiomatic way. (I know Ruby best because I often use it both at work and during my leisure time. It's one of my favorite languages.) However, my objective is just to show that, with some tweaking, you can integrate Spring Security into your projects that are written in a language other than Java.

You'll see that the support for Spring Security in other JVM languages is sometimes straightforward and comprehensive (like when using Groovy and its Grails framework); at other times, you might need to roll your own integration solution to support it. In the upcoming sections, I give a brief overview of how to integrate Spring Security into what I consider to be the three major JVM languages other than Java: Groovy, JRuby, and Scala.

Remember that ultimately Spring Security is no more than a set of Java libraries built on top of Spring Framework, which allows you to plug authentication/authorization security mechanisms into your applications. So it should be possible to integrate it into any Java (Java JVM) application you have. Of course, not all functionality will apply to any application. For example, the filter chain won't make sense in a non-web application.

Spring Security and Ruby (JRuby)

Ruby is definitely one of my favorites languages, and I spend a lot of time working with it. It combines great syntax with great language constructs, and it's a pleasure to work with. Ruby is an object-oriented dynamic language with a focus on productivity, concision, and simplicity.

The standard Ruby interpreter is written in C and was known as *MRI Ruby* until version 1.9. From version 1.9 forward, the official interpreter is known as the *YARV interpreter*.

Ruby is an incredibly popular language, and most of its popularity stems from the widespread use of its incredible web framework Ruby on Rails (RoR), also known simply as *Rails*.

Rails is an MVC framework that places great emphasis on convention over configuration practices. It's a very productive framework you can use to develop simple web applications in a fast and easy way if you follow the conventions enforced by the framework.

I won't give an in-depth explanation of either Ruby or Rails because that would be beyond the scope of this book. I also assume that if you're reading this section, you probably know about them and just want to learn how to integrate Spring Security into them, or to learn if it's at all possible. However, I'll try to give small explanations of Ruby concepts when I use them in the examples.

JRuby is a fully functional implementation of the Ruby programming language written in Java. You can use it to run Ruby programs inside a Java virtual machine and interact with your other JVM languages—mainly, of course, Java.

What I'll show in this section is a simple tutorial for integrating Spring Security into a Rails application and for deciding if it's even worth doing.

In the JRuby case, there is no plugin like the one you find in Grails. In fact, the integration between Java and Ruby is not as smooth as the integration between Java and Groovy. Basically, we'll have to roll our own implementation to make the integration work.

First things first. Let's install JRuby. I'll use the simple installation here, just downloading a file, uncompressing it, and adding its executables to the path. Following is the procedure I use on my Mac (which should be similar to other operating systems):

1. Go to the directory you want to install JRuby in.
2. Download the file `wget http://jruby.org.s3.amazonaws.com/downloads/1.7.0.RC1/jruby-bin-1.7.0.RC1.tar.gz`.
3. Uncompress it using the following: `tar zxvf jruby-bin-1.7.0.RC1.tar.gz`.
4. Go inside the new directory, and add the bin directory to your path in the `.bash_profile`: `export PATH=$PATH:$PWD/bin`.
5. You should have access to JRuby now. Execute `jruby -v`, and you should get the version of JRuby you just installed.
6. Next, let's install Rails: `jruby -S gem install rails`.
7. Now let's create our Rails application. As in the rest of the book, this will be an application with very little functionality just to show you how to use Spring Security. Let's call this application simply *demo*. From a directory of your choice, write the following command: `rails new demo`.

We now have a new basic Rails application in the system. You can run it by going to the directory `demo` that you just created from the command line and entering `rails s`. That command will start a WEBrick server and run the application on that server. If you are following the example, you should be able to visit `http://localhost:3000` in your computer and access the default Rails application home page.

In the process of creating the Rails application, a lot of infrastructure code and a well-defined directory structure was generated for us. If you take a look at the `demo` directory (which is the root of your application), you'll see what I mean.

Anyway, as I said before, I won't go into any depth examining the Rails framework or Ruby itself. I'll simply show with a rudimentary example how you could approach integrating Spring Security into a Ruby on Rails application. I am assuming you already know Ruby and Rails.

Web-Layer Security in Rails

When we ran our Rails application in the previous section, we ran it with a Ruby server (WEbrick). This server doesn't know anything about Java Servlets, so you might guess that it's not possible to run Spring Security web-layer security with this server—and you would be correct. What we need to do is run our Rails application in a standard Java web container, and that is what we'll do next.

First, we install *warbler*, a gem you use to create standard WAR files from our Rails application. To install warbler, use the command `jruby -S gem install warbler`. After warbler is installed, you can execute `jruby -S warble` in the root directory of our application and it will create `demo.war`.

That's good, but let's not deploy it just yet. Let's add some functionality to it first, and then let's add Spring Security–level security.

We'll add two simple routes to our application. One will return the string `SECURED` and will be accessible only to logged-in users. The other one will return the string `UNSECURED` and will be available to any user.

Let's create a pair of controllers: one for admin users and one for standard users. We'll call these controllers simply `AdminsController` and `StandardsController`. Type the following two commands in the root of our `demo` application to generate them:

```
rails g controller admins
rails g controller standards
```

The execution of those commands generates output describing the artifacts that got generated. We'll then edit those controllers to look like Listing 9-21 and Listing 9-22. You can find the controllers in the standard Rails location in the `app/controllers` directory.

Listing 9-21. AdminsController with a secured action

```
class AdminsController < ApplicationController
  def secured
    render :text => "This is top secret code"
  end
end
```

Listing 9-22. StandardsController with an unsecured action

```
class StandardsController < ApplicationController
  def unsecured
    render :text => "Anybody can read this meaningless message"
  end
end
```

The next thing you need to do is to copy all the Java libraries you need to use into the `lib` directory of our Rails application:

```
aopalliance-1.0.jar
commons-codec-1.3.jar
commons-logging-1.1.1.jar
javax.servlet-api-3.0.1.jar
```

```
spring-aop-3.0.6.RELEASE.jar
spring-asm-3.0.6.RELEASE.jar
spring-beans-3.0.6.RELEASE.jar
spring-context-3.0.6.RELEASE.jar
spring-core-3.0.6.RELEASE.jar
spring-expression-3.0.6.RELEASE.jar
spring-jdbc-3.0.6.RELEASE.jar
spring-security-config-3.1.0.RELEASE.jar
spring-security-core-3.1.0.RELEASE.jar
spring-security-crypto-3.1.0.RELEASE.jar
spring-security-web-3.1.0.RELEASE.jar
spring-tx-3.0.6.RELEASE.jar
spring-web-3.0.6.RELEASE.jar
```

The next step is to enable our Rails application to be run in a standard Java web server, such as Tomcat. For that, we'll use the gem *warbler*. Warbler is a JRuby exclusive gem you use to convert different kinds of Ruby applications into standard Java packaging artifacts, such as `Jar` and `War` files. In our case, we'll obviously be creating a `War` file.

Warbler uses internally the gem `jruby-rack` and packs it into the `War`-based application. This gem is the core of the integration between Ruby Rack–based applications (such as Rails applications) and Java web servlet–based applications, which is what we need.

`Jruby-rack` works as a translation layer. It first initializes the Ruby part of the application with a Servlet listener (`org.jruby.rack.rails.RailsServletContextListener`), and then, on each request, it intercepts the calls to the server with a servlet filter, `org.jruby.rack.RackFilter`. When this filter gets the `HttpServletRequest`, it will translate these Java based requests into Rack requests that will pass through to the Rails application, as these are the requests Rails will understand as it is built on the Rack model. Of course, there is a lot more detail regarding warbler and `jruby-rack`, but for the moment, the explanations presented here should be enough for you to continue with the example. In Figure 9-7, you can see a graphical illustration of how `jruby-rack` works.

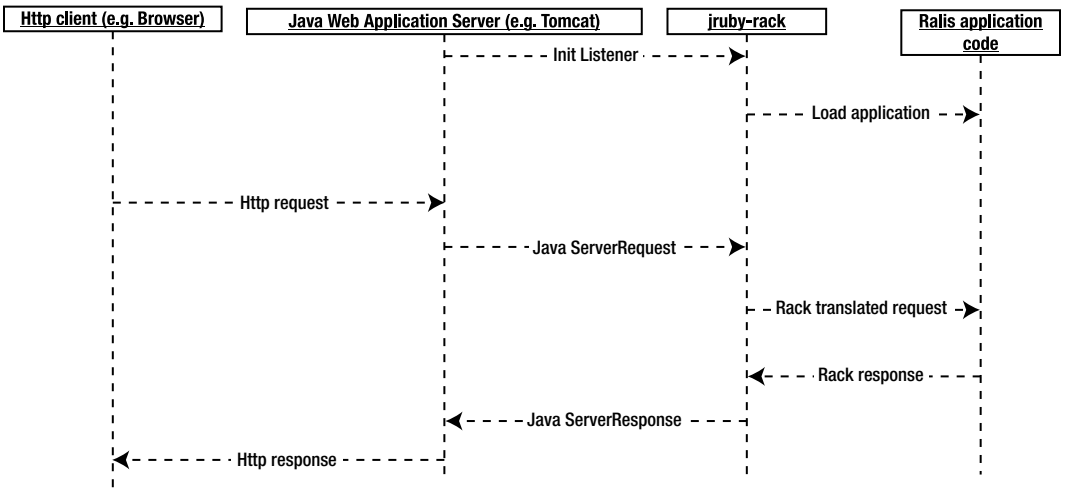


Figure 9-7. *A jruby-rack mapping of the Java and Ruby worlds*

To use warbler with Spring Security. you need to change the `web.xml` file that gets automatically generated by it. To do this, you need to copy the `web.xml.erb` that comes with warbler (which resides in the gem installation directory of Ruby and which is `~/.rvm/gems/jruby-1.6.7/gems/warbler-1.3.6/web.xml.erb` on my computer) into the

config directory of our demo application. Then you need to edit it and make it look like Listing 9-23. By doing this, you are including the Spring Security configuration in our web.xml file as you did in previous chapters.

Next, copy the applicationContext-security.xml file from Listing 9-24 into the root of our demo application. Here we have our familiar Spring Security configuration from Chapter 2, with a few modifications. We are simply securing the URL /admin/* for members of the Scarvarez family.

Then you need to edit warbler's configuration file. To do that, you execute the command "jruby -S warble config" in the root of our application. That execution generates a warble.rb file inside the config directory of the application. Make sure that that file looks like Listing 9-25. In the file, you are ensuring JRuby will be compatible with Ruby 1.9 and that the Spring configuration file will be included in the WEB-INF directory in the generated War file when warbler builds this file.

That's all you need, so let's create the War file again. From the root directory of our application, execute JRUBY_OPTS=--1.9 warble. (This command doesn't work in windows. You should execute just warble and set the JRUBY_OPTS variable separately.) That generates the War file with our Rails application embedded on it. Deploy it to your web application server. (I deployed it to Tomcat 7.)

After it is deployed, you can visit the corresponding URLs. On my computer, the behavior is as follows: when I visit the URL http://localhost:8080/demo/standard/message, I get the message on the screen "Anybody can read this meaningless message."

However, if I visit the URL http://localhost:8080/demo/admin/message, the familiar login page is shown. After I log in with the username **car** and the password **scarvarez**, I see the following message on the page: "This is top secret code." This is the behavior we expected. We have secured our Rails application with basic web-layer security.

Listing 9-23. The web.xml.erb in the config folder, including configuration for Spring Security

```
<!DOCTYPE web-app PUBLIC
  "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
  "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
<% webxml.context_params.each do |k,v| %>
<context-param>
<param-name><%= k %></param-name>
<param-value><%= v %></param-value>
</context-param>
<% end %>
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
           /WEB-INF/applicationContext-security.xml
        </param-value>
    </context-param>

<listener>
    <listener-class>
         org.springframework.web.context.ContextLoaderListener
    </listener-class>
    </listener>
    <listener>
<listener-class><%= webxml.servlet_context_listener %></listener-class>
</listener>

    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
```

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

<filter>
   <filter-name>RackFilter</filter-name>
   <filter-class>org.jruby.rack.RackFilter</filter-class>
</filter>

<filter-mapping>
   <filter-name>springSecurityFilterChain</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
   <filter-name>RackFilter</filter-name>
   <url-pattern>/*</url-pattern>
</filter-mapping>

<% if webxml.jndi then [webxml.jndi].flatten.each do |jndi| %>
<resource-ref>
  <res-ref-name><%= jndi %></res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
</resource-ref>
<% end; end %>
</web-app>
```

Listing 9-24. The applicationContext-security.xml security for /admin/* URLs

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd

    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd">
    <security:http auto-config="true">
        <security:intercept-url pattern="/admin/*" access="ROLE_SCARVAREZ_MEMBER" />
    </security:http>
    <security:authentication-manager>
        <security:authentication-provider>
            <security:user-service>
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="car" password="scarvarez" />
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="mon" password="scarvarez" />
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="bea" password="scarvarez" />
                <security:user authorities="ROLE_SCARVAREZ_MEMBER" name="andr" password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans>
```

Listing 9-25. The warble.rb configuration file

```
Warbler::Config.new do |config|
   config.dirs = %w(app config lib log vendor tmp)
   config.includes = FileList["db"]
   config.webinf_files += FileList["applicationContext-security.xml"]
   config.webxml.jruby.compat.version = "1.9"
end
```

Spring Security, Groovy, and Grails

Groovy is one of the strongest contenders for the number one spot in the non-Java JVM language space. It has a great community and currently is supported and managed by the SpringSource people.

Groovy is a programming language that tries to combine the power of Java with the elegance and developer-friendly characteristics of some dynamic programming languages—taking ideas mainly from well-known and much-loved languages such as Python and Ruby. Groovy successfully creates an environment where different kinds of programmers, with some practice, can feel at home (experienced Java developers and Ruby developers, for example).

Groovy has many advantages when you compare it with other JVM languages—the main ones being, in my opinion, the easy transition from Java to Groovy and the interoperability between the two languages.

With regard to the transition part, it is easy to take a Java program and make it a Groovy program. As a matter of fact, you don't have to do anything. A Java program is already a valid Groovy program. This is good from one point of view but bad from another. Even when you can compile a Java program as a Groovy one, it doesn't make sense. Ultimately, you are using Groovy to take advantage of the great features it has compared to Java. Those great features include a lot of metaprogramming techniques, new powerful constructs like closures, faster development cycles, and a clearer and more concise and developer-friendly syntax.

Grails is a web framework written in Groovy and intended to be used with Groovy. It is built on top of the Spring Framework and is heavily influenced by Ruby on Rails. Grails is an attempt to make a friendlier and lighter framework in a JVM-based language that can be picked up quickly. Also, it offers better and more concise ways to develop applications. At the same time, it is built upon some of the strongest Java libraries and frameworks, such as Spring and Hibernate.

Using Grails to Secure the Web Layer with URL Rules

Grails and Spring Security integration is incredibly simple because Spring Security is the default security solution for Grails applications and a comprehensive plugin exists to support it. Here, we'll create a simple Grails application, and I'll show you how to secure it with Spring Security. As usual, I'll introduce this application in a step-by-step process.

Download and install the latest version of Grails if you still don't have it. You can download it from its homepage at http://grails.org/. I am running this example with version 2.1.1 of Grails.

To install it, simply unzip the downloaded .zip file and then set the environment variable `GRAILS_HOME` to point to the new expanded directory. Also, add to your `PATH` environment variable the path `GRAILS_HOME/bin`.

At the command line, go to any directory where you want to create the application and execute `grails create-app demo-grails` to create the application.

Next, let's generate a couple of controllers: a secured controller and an unsecured controller. Execute `grails create-controller secured` inside the generated application directory. And then execute `grails create-controller unsecured`.

Now we need to create an action in each controller and make them look like Listing 9-26 and Listing 9-27. These files are located in the directory `grails-app/controllers/demo/grails` of your application.

We could run the application now with `grails run-app` and visit either `http://localhost:8080/demo-grails/secured/message` or `http://localhost:8080/demo-grails/unsecured/message`, and you should be able to access both URLs without a problem. The next logical step is to secure the secured URL.

As I said before, Grails is built on top of Spring, so it's only logical that an integration with Spring Security should be straightforward. Grails is built in a clever modular way so that you can add functionality in the form of plugins. Let's install the Spring Security plugin by executing `grails install-plugin spring-security-core` (which, in my current version, installs the version 1.2.7.3). Then execute the command "`grails s2-quickstart demo.security User Role`" to generate the needed models to support users and roles in the application.

Next, we'll create a couple of test users to try out security. Open the file `BootStrap.groovy` (in the `grails-app/conf` directory), and make it look like the coding in Listing 9-28. Here you are creating two users: one with role `ROLE_ADMIN`, and the other one with the role `ROLE_USER`. You can see that this code uses the classes generated in the previous step, `User` and `Role`.

As with previous examples, we'll secure certain URLs so that they are accessible only for users with a specific role. To do this in Grails, you need to add the code from Listing 9-29 to the end of the file `Config.groovy` (which resides in the directory `grails-app/conf`). The code is self-explanatory. It simply tells Grails to use an intercept URL map for security. This map defines URL paths with wildcards (in `Ant`-style syntax, I already covered before for standard Java rules) and the roles that are allowed to access such URLs.

Run the application by executing the command `grails run-app` in the root directory. Now, if you try to access the URLs `http://localhost:8080/demo-grails/unsecured/message` or `http://localhost:8080/demo-grails/secured/message`, you'll get redirected to a login page. If you log in with the username **carlo** and the password **password**, you'll be granted access to the secured URL and not the unsecured one. However, if you log in with the username **monica** and the password **password**", you'll be granted access to `http://localhost:8080/demo-grails/unsecured/message` but you'll get an "access denied" error when trying to access `http://localhost:8080/demo-grails/secured/message`. You can see that you have properly secured access to the URLs.

The first thing that is hard to see from this example is that you are actually using Spring Security. I say this because you haven't defined any filters, any authentication managers, any voters, any user service, or any authentication provider. Actually, you haven't defined any Spring bean at all. You could easily assume that something else is used under the covers because there is nothing specific to Spring Security here. This is a good thing, and one of the nice features of Grails. The use of plugins gives you sensible common defaults and leaves you only with the responsibility of defining the things that are exclusive to your business problem. In the case of security, you need to define your users, their passwords, and the roles and access permissions for your application.

Listing 9-26. SecuredController with a secured message in the Grails application

```
package demo.grails

class SecuredController {

    def message() {
        render "Incredibly confidential message"
    }
}
```

Listing 9-27. UnsecuredController with an unsecured message in the Grails application

```
package demo.grails

class UnsecuredController {

    def message() {
        render "message for everyone"
    }
}
```

Listing 9-28. BootStrap.groovy setting up a couple of test users with security roles assigned to them

```
import demo.security.Role
import demo.security.User
import demo.security.UserRole

class BootStrap {

    def init = { servletContext ->
        def adminRole = new Role(authority: 'ROLE_ADMIN').save(flush: true)
        def userRole = new Role(authority: 'ROLE_USER').save(flush: true)

        def testAdmin = new User(username: 'carlo', enabled: true, password: 'password')
        testAdmin.save(flush: true)
        def testUser = new User(username: 'monica', enabled: true, password: 'password')
        testUser.save(flush: true)

        UserRole.create testAdmin, adminRole, true
        UserRole.create testUser, userRole, false

    }
    def destroy = {
    }
}
```

Listing 9-29. An excerpt from Config.groovy, where we add the URLs that need to be secured

```
grails.plugins.springsecurity.securityConfigType = "InterceptUrlMap"
grails.plugins.springsecurity.interceptUrlMap = [
    '/secured/**':    ['ROLE_ADMIN'],
    '/unsecured/**':   ['ROLE_USER'],
]
```

Grails' Spring Security plugin gives you more access to the functionality offered by Spring Security and, of course, it also allows you to customize it by overriding the defaults. One of the things it supports is the use of SpEL for access rules. To test this in the context of our example, and with the simplest security SpEL expression I can think of, simply replace the security section I introduced in the file `Config.groovy` (from Listing 9-29 which we just used before) with the content from Listing 9-30. In that listing, as you can see, we are using the security expression `"hasRole"`, which you already studied in previous chapters. Note that this is a simple example with `"hasRole"`, but here you have access to the full suite of expressions offered by Spring Security SpEL support. For example, you could use `"authentication.name == 'carlo' "` as an expression instead of `"hasRole"`.

Listing 9-30. An excerpt from Config.groovy, which uses SpEL instead of simple roles

```
grails.plugins.springsecurity.securityConfigType = "InterceptUrlMap"
grails.plugins.springsecurity.interceptUrlMap = [
    '/secured/**':    ["hasRole('ROLE_ADMIN')"],
    '/unsecured/**':   ["hasRole('ROLE_USER')"],
]
```

Using Grails Security at the Method Level

Grails' Spring Security plugin supports method-level security the same way as it does with Java. This means you can put `@grails.plugins.springsecurity.Secured` annotations (or the standard Spring Security `@Secured` annotation as well) in our controller classes and Spring Security will make sure they are secured according to those annotations. Doing that is very simple. Let's keep working on the application from last section. First, replace the current content of the file `Config.groovy` (in the `grails-app/conf` directory) from the line that reads `grails.plugins.springsecurity.securityConfigType = "InterceptUrlMap"` onward with the simple line `grails.plugins.springsecurity.securityConfigType = "Annotation"`. Then replace the content of class `SecuredController` (in the `grails-app/controllers/demo/grails` directory) with the content of Listing 9-31 and the content of class `UnsecuredController` (in the `grails-app/controllers/demo/grails directory`) with the content of Listing 9-32. After you do this, if you restart the application, you should get the same access constraints that you got in the previous section when you secured the URLs.

Listing 9-31. SecuredController with the @Secured annotation

```
package demo.grails

import grails.plugins.springsecurity.Secured;

class SecuredController {

    @Secured(["ROLE_ADMIN"])
    def message() {
        render "Incredibly confidential message"
    }
}
```

Listing 9-32. UnsecuredController with the @Secured annotation

```
package demo.grails

import grails.plugins.springsecurity.Secured;

class UnsecuredController {

    @Secured(["ROLE_USER"])
    def message() {
        render "message for everyone"
    }
}
```

Download from Wow! eBook <www.wowebook.com>

In these last two sections, I just scratched the surface of the functionality available using Grails' Spring Security plugin. You can do virtually everything you can do using the standard Java support. For a comprehensive guide to using the Grails plugin, take a look at its official page at http://grails-plugins.github.com/grails-spring-security-core/docs/manual/.

Spring Security and Scala

Scala is probably the strongest language (in the sense of number of adopters and the liveliness of the community around it) running on top of the Java Virtual Machine apart from Java itself. (Scala also has a version that runs on the .NET platform.) Scala is a very powerful, general-purpose programming language that tries to merge the best of the object-oriented and functional programming paradigms.

Scala is a language that aims to provide a concise and elegant alternative to the world of enterprise Java programming, while keeping the type safety of Java. As I said, it comes with functional programming features built in, adding a whole new layer of power for the seasoned object-oriented programmer that increases productivity.

I am not an expert in Scala or functional programming (far from it as I have just recently started to look at them) but I will give you the core definitions and the main characteristics of both and show you how to use them with Spring and Spring Security.

Functional programming is a paradigm in which programs are composed of functions that receive inputs and produce outputs, while also avoiding the use of state and mutability. This is in clear contrast to object-oriented programming, where the main abstraction is the object and its internal state, and mutability is a very common thing. By not allowing mutability and state, functional programming presents itself as a good alternative for programming concurrent programs, because programmers don't need to worry about synchronization between concurrent processes or threads.

In Scala, functions can be passed around (as function parameters or function return values) as simple values. In this sense, they behave pretty much like any other simple value (for example, a string), and this is an important concept in a functional programming language. In Scala, functions are first-class objects.

According to the literature, Scala's name comes from the words *scalable language*. This implies that Scala supports object-oriented and functional programming paradigms and that it is suitable for simple scripting tasks or full enterprise applications. Combining this capability with an elegant and concise syntax makes the language scalable in terms of the number of domains and uses that it can address.

Let's install Scala now. The easiest thing to do is just go to its home page (http://www.scala-lang.org/downloads/distrib/files/scala-2.9.2.tgz) and download the latest version (2.9.2 at the time of this writing). Uncompress the downloaded file in your directory of choice (which I will call `YOUR_SCALA_HOME` from now on). Then if you are using Mac or Linux with bash shell, copy the two lines from Listing 9-33 to the file `~/.bash_profile` and then execute the command `"source ~/.bash_profile"` so that it loads the file again. You have now set up Scala to be used from the command line.

Listing 9-33. Adding Scala to your .bash_profile

```
export SCALA_HOME=YOUR_SCALA_HOME
export PATH=$PATH: SCALA_HOME /bin
```

You can try it a bit from the command line. To do that, execute the command `scala`. You'll be presented with a prompt. Figure 9-8 shows some interaction with this command-line REPL (read-eval-print loop) interpreter.

```
Welcome to Scala version 2.9.2 (Java HotSpot(TM) 64-Bit Server VM, Java 1.7.0_05).
Type in expressions to have them evaluated.
Type :help for more information.

scala> 4+5
res0: Int = 9

scala> println("Hello world")
Hello world

scala> val entero = 5
entero: Int = 5

scala> entero + 1
res2: Int = 6

scala> def suma(val1:Int,val2:Int):Int ={val1+val2}
suma: (val1: Int, val2: Int)Int

scala> suma(3,3)
res3: Int = 6

scala>
```

Figure 9-8. *Some interaction with the Scala command-line interpreter*

The preceding figure shows a simple set of commands in the Scala REPL interpreter. First, you perform a simple `sum` operation using two integer values. Then you call the built-in function `println`. Then you define the value `entero` of type `Int` (implicit) and assign it the value `5`. The next line defines a new function called `suma` that receives two `Int` parameters and returns an `Int` value.

■ **Note** Scala makes a distinction between *values* and *variables*. Values are like constants in other programming languages, and they are values that can't change. (They are immutable.) If you try to modify the *entero* value shown in Figure 9-8, you'll get an error message such as "error: reassignment to val." Values are defined with the *val* keyword. Variables, on the other hand, are much like variables in any other language, and you can reassign values to them whenever you want. They are defined with the *var* keyword, such as *var entero2 = 3*.

OK, that was your introduction to Scala! It was a very simple introduction, I know. However, as I said, covering the language in depth is outside the scope of this book and there are a lot of great books dedicated to the topic. I'm assuming here that you probably know the language better than me and you are just interested in how to use Spring Security with it.

The next thing we'll do is create the project by combining Scala with Spring and Spring Security.

Again, we'll use our old friend Maven to create and manage the project. From the command line, in a directory of your choice, execute the command shown in Listing 9-34 to create the new project.

Listing 9-34. Creating the Scala project

```
mvn archetype:generate -DgroupId=com.apress.pss\
 -DartifactId=scala-example\
 -DarchetypeArtifactId=maven-archetype-webapp
```

In the generated `pom.xml` file, add the Scala dependency shown in Listing 9-35.

Listing 9-35. Scala Maven dependency

```
<dependency>
    <groupId>org.scala-tools</groupId>
    <artifactId>maven-scala-plugin</artifactId>
    <version>2.15.2</version>
</dependency>
```

Next, of course, you need to add the Spring dependencies to the `pom.xml` file. You should know how to do this by now. You also need to add the Scala plugin in the plugin sections. In the end, your `pom.xml` file should look like Listing 9-36.

Listing 9-36. The pom.xml file in the Scala Maven project

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.apress.pss</groupId>
    <artifactId>scala-example</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>scala-example Maven Webapp</name>
    <url>http://maven.apache.org</url>
    <repositories>
        <repository>
            <id>scala-tools.org</id>
            <name>Scala-tools Maven2 Repository</name>
            <url>http://scala-tools.org/repo-releases</url>
        </repository>
    </repositories>
    <pluginRepositories>
        <pluginRepository>
            <id>scala-tools.org</id>
            <name>Scala-tools Maven2 Repository</name>
            <url>http://scala-tools.org/repo-releases</url>
        </pluginRepository>
    </pluginRepositories>
    <dependencies>
        <dependency>
            <groupId>org.scala-tools</groupId>
            <artifactId>maven-scala-plugin</artifactId>
            <version>2.15.2</version>
        </dependency>
        <dependency>
            <groupId>org.scala-lang</groupId>
            <artifactId>scala-library</artifactId>
            <version>2.9.2</version>
        </dependency>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
```

```
            <version>3.0.1</version>
        </dependency>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.5</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-core</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-config</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-web</artifactId>
            <version>3.1.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.1.1</version>
        </dependency>
        <dependency>
            <groupId>commons-codec</groupId>
            <artifactId>commons-codec</artifactId>
            <version>1.3</version>
        </dependency>
    </dependencies>
    <build>
        <finalName>scala-example</finalName>
        <sourceDirectory>src/main/scala</sourceDirectory>
        <testSourceDirectory>src/test/scala</testSourceDirectory>
        <plugins>
            <plugin>
                <groupId>org.scala-tools</groupId>
                <artifactId>maven-scala-plugin</artifactId>
                <executions>
                    <execution>
                        <id>scala-compile-first</id>
                        <phase>process-resources</phase>
                        <goals>
                            <goal>add-source</goal>
                            <goal>compile</goal>
                        </goals>
                    </execution>
```

```
                <execution>
                    <id>scala-test-compile</id>
                    <phase>process-test-resources</phase>
                    <goals>
                        <goal>testCompile</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
        <plugin>
            <groupId>org.mortbay.jetty</groupId>
            <artifactId>jetty-maven-plugin</artifactId>
            <version>8.1.1.v20120215</version>
            <configuration>
                <connectors>
                    <connector implementation="org.eclipse.jetty.server.nio.
                    SelectChannelConnector">
                        <port>8080</port>
                        <maxIdleTime>60000</maxIdleTime>
                    </connector>
                </connectors>
            </configuration>
        </plugin>
    </plugins>
  </build>
</project>
```

This will be a simple web application with Spring MVC, and it will have a very simple Service layer. The big difference from applications you saw in other chapters is that both the Controller layer and the Service layer will be written in Scala instead of Java. I'll show you how to add security to the method level, but as you'll see it is almost the same as with Java. Let's start with the code for the controller. In the package `com.apress.pss.scala.web` in a source folder with the path `src/main/scala`, create the `ScalaController` class shown in Listing 9-37.

Listing 9-37. ScalaController

```
package com.apress.pss.scala.web;
import org.springframework.beans.factory.annotation.Autowired
import org.springframework.web.bind.annotation.RequestMapping
import org.springframework.web.bind.annotation.RequestMethod
import org.springframework.stereotype.Controller
import com.apress.pss.scala.service.ScalaServiceFacade
import javax.servlet.http.HttpServletResponse
import javax.servlet.http.HttpServletRequest

@RequestMapping(Array("/enter"))
class ScalaController(service: ScalaServiceFacade) {

   @RequestMapping(value = Array("/scala"), method = Array(RequestMethod.GET))
   def scalaRequest(request:HttpServletRequest, response:HttpServletResponse) = {
    val value = service.scalaService
    response.getWriter().write(value)
   }
}
```

In the controller, we are defining a simple method that returns (writes on the response, actually) whatever the service returns. You can see how we are using the annotation `@RequestMapping` in the example. With Scala, we cannot use a simple string to set the value of array-based annotation values. We need to set a real array with just one element as the value. For the rest, the example is very much like the Java version, with some syntax modifications.

Then, in the package `com.apress.pss.scala.service`, we create the `ScalaService` class shown in Listing 9-38.

Listing 9-38. ScalaService

```
package com.apress.pss.scala.service;
import org.springframework.stereotype.Service
import org.springframework.security.access.annotation.Secured

trait ScalaServiceFacade {
    def scalaService: String
}

class ScalaService extends ScalaServiceFacade{
  @Secured(Array("ROLE_USER"))
  def scalaService() = "Service accessed"
}
```

Again, the code in the preceding listing is straightfoward. It is a service class with a simple method that returns a string. Notice how we are using the `@Secured` annotation here. Again, we are using the `Array` function (actually what we are indirectly calling here is the `apply` method of the `Array` companion object which allows us to create a new instance of the Array class) and passing the string that will be the only element of the array. With Scala, we can't use the convenient technique of passing a simple string for this value as we did in the Java version.

Note also the use of the `trait` just before the class definition and then the class extending that trait. A `trait` in scala is somewhat the equivalent of the interfaces in Java, but it has a lot more power. Although not shown in this example, a trait can have fully implemented methods as well as the traditional abstract (definition only) methods typical of Java interfaces, and your class can extend more than one trait 'inheriting' the functionality defined in all of them without needing to implement the already implemented methods.

This technique, also called *mixin*, is in no way exclusive to Scala because other languages also have constructs that fulfil the same purpose. For example, in Ruby you can use modules to achieve more or less the same outcome that you get with Scala traits. Again, I won't go into any details about this. For our example, you can think of a trait simply as a Java interface that you use from the controller to access the service.

Next, we need to make our configuration. By now you should be very familiar with configuring a Spring Security web application. So I will simply show you the files next and won't go into the details of any of them. Note that we are referencing Scala classes just as we used to reference Java classes before in our beans. It is great that interoperability between Java and Scala is so nicely achieved. Listing 9-39 shows the `web.xml` file. Listing 9-40 shows the `applicationContext-security.xml`. Listing 9-41 shows the file `scala-servlet.xml`. All files will live under the `WEB-IF` directory of your application.

Listing 9-39. The web.xml file for the Scala project

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
      http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd" version="3.0">
    <listener>
      <listener-class>
         org.springframework.web.context.ContextLoaderListener
      </listener-class>
    </listener>
```

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

    <servlet>
        <servlet-name>scala</servlet-name>
        <servlet-class> org.springframework.web.servlet.DispatcherServlet </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>scala</servlet-name>
        <url-pattern>/</url-pattern>
    </servlet-mapping>
    <filter>
        <filter-name>springSecurityFilterChain</filter-name>
        <filter-class>
            org.springframework.web.filter.DelegatingFilterProxy
        </filter-class>
    </filter>
    <filter-mapping>
        <filter-name>springSecurityFilterChain</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
</web-app>
```

Listing 9-40. The applicationContext-security.xml for the Scala project

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security="http://www.springframework.org/schema/security"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-security-3.1.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

    <bean id="scalaService" class="com.apress.pss.scala.service.ScalaService"/>

    <security:http auto-config="true" />

    <security:authentication-manager>
        <security:authentication-provider>
```

```
            <security:user-service>
                <security:user name="car" authorities="ROLE_USER"
                    password="scarvarez" />
                <security:user name="mon" authorities="ROLE_ADMIN"
                    password="scarvarez" />
            </security:user-service>
        </security:authentication-provider>
    </security:authentication-manager>
</beans>
```

Listing 9-41. The scala-servlet.xml file with the controller definition

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:security=http://www.springframework.org/schema/security

    xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xsi:schemaLocation="http://www.springframework.org/schema/mvc
    http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd
    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-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-3.1.xsd">

    <mvc:annotation-driven />

    <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name = "prefix" value="/WEB-INF/views/" />
        <property name = "suffix" value=".jsp" />
    </bean>

    <bean id="scalaController" class="com.apress.pss.scala.web.ScalaController">
      <constructor-arg ref="scalaService"/>
    </bean>

</beans>
```

Now we can exercise our project. From the root of the project in the command line, execute the command `mvn clean install jetty:run` to run the application. It should run without any problem.

Next, if you try to access the URL `http://localhost:8080/enter/scala`, you'll be presented with the login screen you have seen so many times before.

If you log in with the username **car** and the password **scarvarez**, you should be able to access the application and see the page shown in Figure 9-9.

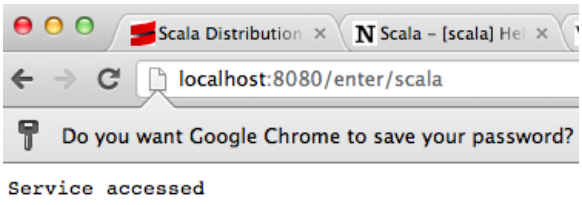


Figure 9-9. *Successfully accessing the secured Scala application*

That's all I am going to say about integrating with Scala. You can see that for a simple application like this it's not really that different than integrating with Java itself. Actually, beyond the syntactic differences, it's probably just about the same. Of course, in this example you are starting with a Spring application and defining the different components (controllers and services) in Scala. It could be more of a challenge to integrate if you are using a Scala-specific framework. All you need to remember is that to secure methods, those methods need to belong to Spring-managed beans. To secure URLs, the URLs need to be accessed in a Servlet-based web application.

Summary

In this chapter, I showed you, at a fairly high level, how to integrate Spring Security into various frameworks and languages. The chosen framework was Spring's own Spring Web Flow, and the illustrative languages chosen were JRuby and Groovy, with their web frameworks Rails and Grails, respectively.

I showed you that integrating Spring Security into these different frameworks and languages can be straightforward (as with Grails) or not so straightforward (as with JRuby or Scala). However, through the JVM, in theory, you could integrate Spring Security into anything you need to secure.

Here is the main takeaway from this chapter, regardless of the frameworks or languages used: Spring Security is, ultimately, just a Java library (a couple of simple `jar` files). You can integrate it into any Java (as in JVM) project that you want. You simply have to remember what you can do with it and use the parts that make sense for your particular problem. For instance, using web-layer security wouldn't make sense in a Swing application, or even in a web application that doesn't use standard servlet filters (such as the Play framework).

Index

Download from Wow! eBook <www.wowebook.com>

A

AbstractSecurityInterceptor, 31
AbstractSecurityInterceptor's beforeInvocation method, 122
Access Control Entry (ACE), 207
Access control lists (ACLs)
  accessing secured objects
    AclEntryVoter(s), 222, 226
    custom AccessDecisionManager, 223
    default Spring Security login page, 225
    deletePost method, ForumController, 224
    deletePost method, ForumServiceImpl, 224
    form.jsp, 224
  cost, 234
  filtering returned objects
    classes and interfaces, 231
    classes participating in post-processing phase, 232
    createPost method, 229
    ExpressionHandler bean, 228
    getPosts method, 227
    @PostFilter annotation, 231
    @PreFilter annotation, 231
    @PreFilter invocations, 232
    PreInvocationAuthorizationAdviceVoter, 227
    steps to execute new configuration, 230
  securing view layer, 233
  security example application
    abstractions, 207
    acl-example-servlet.xml, 220
    applicationContext-acl.xml, 213
    applicationContext-security.xml, 221
    attributes, 208
    BasePermission class, 208
    createAclSchema.sql, 206
    DatabaseSeeder class, 216
    Entity-Relationship (ER) diagram, 209
    form.jsp file, 219
    ForumController entry point, 216
    ForumServiceImp, 217, 220
    Maven dependencies, 210
    Maven jetty plugin in pom.xml, 212
    permission logic, 205
    Post class domain model, 218
    tables in graphical form, 207
    web.xml, 212
AccessDecisionManager, 30, 49, 289
AccessDecisionVoter, 51
AccessDeniedException, 125
AccessOperationsImpl, 135–136
aclAuthorizationStrategy bean, 215
aclCache bean, 215
ACL_CLASS, 208
ACL_ENTRY, 208
acl module, 20
acl_object_identity, 208
aclService bean, 215
ACL_SID, 208
AfterInvocationManager's, 31
always-use-default-target, 176
applicationContext-security.xml file, 24, 137
Application security layer, 2
AspectJ Maven Dependency, 132
AspectJ pointcut expressions, 131
Aspect Oriented Programming (AOP), 14
aspects module, 21
authentication-failure-handler-ref, 176
authentication-failure-url, 176
AuthenticationProvider, 48
AuthenticationProvider and UserDetailsService
  applicationContext-mongodb.xml, 250
  applicationContext-security.xml, 247
  basic web.xml with enabled Spring Security, 247
  definition, 243, 245
  dependencies, 246
  "Hello World" page, 256
  Hello World Servlet, 248

AuthenticationProvider and UserDetailsService (*cont.*)
Jetty Plugin, 246
MongoUserDetailsService, 249, 253
newly created user, 255
relationship, 244
Small Main Class, 255
spring-data-mongodb dependency, 249
UserReadConverter, 251
Authentication-related events, 240
authentication-success-handler-ref, 177
authorityGranters, 186
Authorization-related events, 239

B

BeanDefinitionParser objects, 37
Business service-level security, 111

C

callbackHandlers, 186
cas module, 20
Central authentication service (CAS) authentication
applicationContext-security.xml spring file, 197
configuration changes in SSL elements, 198
demo application login, 193
with different authentication provider, 202
element, 190
pom.xml file
CAS authentication-powered application, 194
CAS war application, 191
process, 200, 202
secured resources, 199
ticket, 201
web.xml file, 196
ChannelSecurityInterceptor, 269
com.apress.pss.terrormovies.access package, 134
config module, 20
ContextLoaderListener, 24
core module, 20
crypto module, 21
Custom Login Form
applicationContext-security.xml, 73
AuthenticationFailureHandler implementation, 76
authentication-failure-url, 75
custom error, 75
DefaultLoginPageGeneratingFilter, 73
default-target-url, 75
JSP file, 74
j_username and j_password, 75
LoginController creation, 74
new login form, 75
new login handler page, 73
spring security, 75
view resolver, 74

D

Database-provided authentication
applicationContext-security.xml file, 155
basic tables creation, 159
with groups, 154
HSQLDB Maven dependency, 158
vs. memory-provided authentication, 153
modified applicationContext-security.xml, 159
pom.xml, 156
Servlet definition, 156
simple database schema, 153
using existing schemas, 162
using groups, 161
web.xml file, 155
Decorator Pattern, 55
DefaultFlowStateSecurityExpression
Handler, 290
default-target-url, 177
DelegatingFilterProxy, 25
Dependency injection (DI), 13
Digest authentication, 78
DispatcherServlet servlet, 284

E

entries_inheriting, 208

F

Filters and filter chain
ANONYMOUS_FILTER, 41
BASIC_AUTH_FILTER, 41
CHANNEL_FILTER, 40
CONCURRENT_SESSION_FILTER, 40
DIGEST_AUTH_FILTER, 41
EXCEPTION_TRANSLATION_FILTER, 41
FILTER_SECURITY_INTERCEPTOR, 41
FORM_LOGIN_FILTER, 40
JAAS_API_SUPPORT_FILTER, 41
LOGIN_PAGE_FILTER, 41
LOGOUT_FILTER, 40
OPENID_FILTER, 41
PRE_AUTH_FILTER, 40
REMEMBER_ME_FILTER, 41
REQUEST_CACHE_FILTER, 41
SECURITY_CONTEXT_FILTER, 40
SERVLET_API_SUPPORT_FILTER, 41
servlet filter, 39
SESSION_MANAGEMENT_FILTER, 41
SWITCH_USER_FILTER, 42
web.xml file, 40
X509_FILTER, 40
FilterSecurityInterceptor, 30
Functional programming, 301

G

Global-method-security, 116
Gradle Wrapper, 19
Grails
    method level, 300
    web layer with URL rules, 297
Groovy, 291, 297. *See also* Grails

H

Hashing algorithms, 4

I

Inversion of Control (IoC), 13

J, K

JAAS authentication
    bean properties, 186
    configuration file, 185
    JaasAuthenticationProvider, 189
    pss_jaas.config file, 186
    RoleGranterFromMap, 186
    SampleLoginModule, 187
Java Authentication and Authorization Service (JAAS), 8
Java Certification Path API (CertPath), 8
Java Cryptographic Extensions (JCE), 8
Java Cryptography Architecture (JCA), 8
Java Secure Socket Extension (JSSE), 8
JdbcMutableAclService, 234
Jetty server, 25
JRuby, 291
jsr250 Maven dependency, 121

L

LDAP authentication
    applicationContext-security.xml file, 168
    attributes, 163
    context entry, 165
    dNSDomain, 166
    entry, 163
    group-role-attribute, 171
    LdapAuthenticationProvider at work, 170
    LDAP hierarchy, 168
    LDAP server connection, 164
    LDIF file, 166
    LDIF file with Apache Directory Studio, 168
    local ApacheDS server connection, 165
    <password-compare> element, 171
    password encoders, 171
    role-prefix, 171
    Spring Security LDAP dependency, 168
    user-context-mapper-ref, 171
    values, 163
ldap module, 20
loginConfig, 186
loginContextName, 186
login-page-url, 177
login-processing-url, 177

M

MethodSecurityEvaluationContext, 123
MethodSecurityInterceptor, 30
Model-View-Controller (MVC) framework, 273
    Admin controller, 62
    admin user and roles, 63
    AnonymousAuthenticationFilter, 68
    Authentication filter, 71
    BasicAuthenticationFilter, 67
    characteristics, 61
    Curl command, 63
    DefaultLoginPageGeneratingFilter, 67
    ExceptionTranslationFilter, 68
    FilterSecurityInterceptor, 69
    LogoutFilter, 67
    movie creation, 63
    RequestCacheAwareFilter, 67
    RequestMapping annotation, 62
    SecurityContextPersistenceFilter, 66
    Servlet listener, 65
    servlet-name value, 61
    SessionManagementFilter, 68
    terrormovies-servlet.xml, 64
    URL access, 65
    WEB-INF/terrormovies-servlet.xml file, 61
    web.xml snippets, 61
MongoDB, 249
moviesService Bean, 133
MoviesServiceImpl class, 131–132
MRI Ruby, 292
mutual authentication, 178
myOpenID, 172

N

Network security layer, 1

O

object_id_class, 208
Object Identity, 207
object_id_identity, 208
One-way encryption, 4
opened module, 20

OpenID authentication
auto-register functionality, 174
configuration file for Spring Security application, 173
login form, 174
Maven dependencies, 173
MyOpenID site, 175
Spring Security OpenID namespace, 176
workflow, 175
OpenIDAuthenticationToken, 46
Operating system layer, 1
org.springframework.security.acls.jdbc.BasicLookupStrategy class, 235
owner_sid, 208

P, Q

param contextConfigLocation, 24
parent_object, 208
Password encryption
custom security filter
applicationContext-security.xml, 264
error page, 265
UserAgentFilter, 263
handling errors and entry points
AuthenticationEntryPoint Implementation, 266
CookieAccessDeniedHandler, 268
ExceptionTranslationFilter and AuthenticationEntryPoint relationship, 266
Spring Configuration, 267
PreAuthenticatedAuthenticationToken, 45
Public key cryptography, 6

R

Rails, 292
AdminsController, 293
applicationContext-security.xml file, 295
applicationContext-security.xml security, 296
Java libraries, 293
Java web container, 293
jruby-rack, 294
StandardsController, 293
warble.rb configuration file, 297
warbler, 293
WEB-INF directory, 295
web.xml.erb, 295
Remember-me authentication, 80
AffirmativeBased access-decision manager, 82
Amazon.com, 80
AuthenticatedVoter, 82
Authentication object implementation, 82
autoLogin method, 81
cookie, 80
PersistentTokenBasedRememberMeServices, 82
RememberMeServices implementation, 81
RoleVoter, 82
<security:intercept-url/> element, 81
UnanimousBased access-decision manager, 82
UsernamePasswordAuthenticationFilter, 80
remoting module, 20
@RequestMapping annotations, 112
@RolesAllowed annotation, 120
Ruby, 292
Ruby on Rails (RoR), 292. *See also* Rails
RunAsUserToken, 46

S

Scala, 291
application access, 309
applicationContext-security.xml, 307
.bash_profile, 301
command-line interpreter, 302
functional programming, 301
Maven dependency, 303
mixin, 306
pom.xml file, 303
project creation, 302
REPL interpreter, 302
scalable language, 301
ScalaController class, 305
ScalaService class, 306
scala-servlet.xml file, 308
@Secured annotation, 306
Service layer, 305
values and variables, 302
web.xml file, 306
@Secured annotation, 116
SecuredController, 300
Secure Sockets Layer (SSL), 178
Security
application layer
ACLs, 4
authentication, 2, 4
authorization, 3–4
cross-site scripting, 7
denial, service attacks, 7
identity management, 7
Java options, 8
Network security layer, 1
operating system layer, 1
output sanitation, 7
secured connections, 7
sensitive data protection, 7
SQL injection, 7
SecurityExpressionHandler, 289
Security identity (SID), 207
Security interceptor
AbstractSecurityInterceptor, 31
AccessDecisionManager, 30
AfterInvocationManager's, 31

FilterSecurityInterceptor, 30
MethodSecurityInterceptor, 30
preprocessing and postprocessing step, 30
UML class diagram, 30
sernamePasswordAuthenticationToken., 45
Service layer security
access, 120
AccessOperationsImpl, 135–136
AdminController, 112
applicationContext-security.xml, 114–115
CGLIB, 113
class cast exception, 116
class-level and method-level annotations, 116
Global-method-security, 116
login form, 117
MVC mechanism, 112
New AdminController hierarchy, 112
standard JDK proxies, 112
applicationContext-security.xml file, standalone application, 137
AspectJ AOP
applicationContext-security.xml, 148
AspectJ Security Aspect, 142–143
Controller TheController, 149
methodA debugging, 150
methodB debugging, 150–151
page access, 150
pom.xml file for AspectJ example, 143
Service Service.java, 149
weaving, 141
web.xml, 147
business service-level security, 111
command outputs and exceptions, 139
FilterSecurityInterceptor, 117
main class, 134–135
MethodSecurityInterceptor, 117
MovieController functionality, 119
MoviesService and MoviesServiceImpl, 118
package com.apress.pss.terrormovies.access, 133
@RolesAllowed annotation, 120
SpEL expression (*see* SpEL expression)
toString Method, 120
web-based authentication, 140
web-level security, 111
XML
AspectJ Maven Dependency, 132
AspectJ pointcut expressions, 131
moviesService Bean, 133
MoviesServiceImpl class, 132
Service tickets, 190
Servlet Filters, 111
Session-related events, 242
SpEL expressions
@PostAuthorize annotation, 123
afterInvocation, 124
MethodSecurityInterceptor wrapping, 124
movie access, 125
MoviesServiceImpl class, 124
@PostFilter annotation
admin user, 130
allMovies.jsp, 129
DefaultMethodSecurityExpressionHandl, 129
IllegalArgumentException, 129
MovieController method, 129
MoviesServiceImpl class, 129
standard user, 130
@PreAuthorize annotation, 122
@PreFilter annotation
error page, 126
filterObject value, 126
input box, 126
MovieController, 125, 127–128
MoviesServiceImpl method, 125, 128–129
movie storage, 126
newMovies.jsp, 125, 127
pom.xml file, 126
security constraints, 121
Spring Framework
Aspect Oriented Programming, 14
dependency injection, 13
Spring Security
ACLs (*see* Access control lists (ACLs))
Active Directory, 10
application process, 24
databases, 10
definition, 9
design and patterns
Decorator Pattern, 55
dependency injection (DI), 56
SRP, 56
strategy pattern, 55
domain model, 10
event system
AuthenticationProvider and UserDetailsService, 243
authentication-related events, 240
authorization-related events, 239
event mechanism, 238
session-related events, 242
100-foot view
AccessDecisionManager, 49
AccessDecisionVoter, 51
ACL, 54
Authentication object, 44
AuthenticationProvider, 48
ConfigAttribute, 42
filters and filter chain (*see* Filters and filter chain)
JSP Taglib, 54
key components, 29
SecurityContext and SecurityContextHolder, 46
security interceptor (*see* Security interceptor)

Spring Security (*cont.*)
UserDetailsService and AuthenticationUserDetailsService, 52
XML namespace (*see* XML namespace)
1,000-foot view, 28
10,000-foot view, 27
Github, 18
Gradle, 19
Grails
method level, 300
web layer with URL rules, 297
Groovy, 291, 297
hiding elements, 11
HTTP status code handling, 11
Java, 10
Java EE Server, 11
JRuby, 291
layered security services, 10
LDAP, 10
Maven dependencies, 21
modules, 20
nonintrusive and declarative application, 11
OpenID, 10
open source software, 10
out-of-the-box integration, 9
password encryption
changing security interceptor, 269
custom security filter, 262
extensions project, 271
handling errors and entry points, 265
New Expression Root and SpEL, 262
non-JDBC AclService, 262
nonvoter AccessDecisionManager, 259
sha-256 Individual Bean, 257
sha-256 Password encoder, 257
User Inserter Main Method, 257
voters in AccessDecisionManager, 257
public/private key certificates, 11
role-based authentication/authorization, 10–11
and Ruby, 292
Scala (*see* Scala)
service layer, 11
Servlet-based web application
"Hello World" message, 17
HelloWorldServlet, 17
Jetty plugin dependency, 16
pom.xml file with Servlet dependencies, 16
source code folder, 19
and Spring, 12
Spring Framework, 9
Aspect Oriented Programming, 14
dependency injection, 13
Spring Web Flow (*see* Spring Web Flow)
Struts 2 (*see* Struts 2)
web application, 10
web-layer security, Rails (*see* Rails)
web project configuration
applicationContext-security.xml, 22
incorrect user name result, 24
listener configuration, 23
Login page, 23
web.xml, 22
springSecurityFilterChain filter, 25, 284
Spring Web Flow
applicationContext-security.xml file, 285
buy.jsp, 287
DispatcherServlet servlet, 284
example-webflow.xml, 285
flow-executor, 288
main.jsp, 286–287
pom.xml file, 281
products-servlet.xml file, 284
product.xml, 286
review.jsp, 287
SecurityFlowExecutionListener, 288
SpEL-based security, 289
springSecurityFilterChain filter, 284
Spring Security Listener bean, 288
web.xml, 283
working, 280
StateExpressionVoter, 289
StateSecurityExpressionRoot, 291
Strategy pattern, 55
Struts 2
applicationContext.xml file, 275–276
application file structure, 275
Java web framework, 273
MVC framework, 273
secured application, 280
secured HelloWorldAction, 279
Spring Security dependencies, pom.xml file, 277
struts.xml file, 275
web.xml with filter, 278
working, 273–274
Symmetric encryption, 5

T

taglibs module, 20
Truststore, 181

U, V

UnsecuredController, 300

W

Warbler, 293–294
Web-level security, 111
web module, 20

Web security
AdminController, 84
Apache Tomcat, 60
applicationContext-security.xml, 86
ConcurrentSessionControlStrategy
applicationContext-security.xml, 101
chroem and firefox, 99
errors, 101
CustomInMemoryUserDetails
Manager class, 88
Custom Login Form (*see* Custom Login Form)
custom User and lastname retrieving, 87
custom user class, 86
different pattern matchers, 101
different user
inMemoryUserServiceWith
CustomUser, 96
MovieController, 98
Movie model class, 97
roles, 95
digest authentication, 78
ExpressionHandler
access denied, 94
age attribute, 93
applicationContext-security.xml, 93
configuration, 91
CustomWebSecurityExpression
Handler, 92
CustomWebSecurityExpression
Root, 93
Hello World page, 60
HTTP authentication, 77
HTTPS channel security, 102
configuration, 102
vs. HTTP, 102
pom.xml plugin section, 103–104
self-signed certificate, 103
working principle, 104
InMemory model, 85
Jetty application, 60
JSP Taglib
authentication, 107
<authorize> security tag, 106
MovieController, 105
output content, 105
security-oriented tags and
attributes, 104
logging out, 83
new Maven web application, 57
pom.xml file, 58–59
remember-me authentication
(*see* Remember-me authentication)
role hierarchies, 108
SecurityContextHolder, 84
SessionFixationProtectionStrategy, 99
Spring expression language
applicationContext-security.xml file, 90
functionality, 89
WebExpressionVoter, 90
WebSecurityExpressionRoot, 90
Spring MVC
Admin controller, 62
admin user and roles, 63
AnonymousAuthenticationFilter, 68
Authentication filter, 71
BasicAuthenticationFilter, 67
characteristics, 61
Curl command, 63
DefaultLoginPageGeneratingFilter, 67
endpoint method, 71
ExceptionTranslationFilter, 68
FilterSecurityInterceptor, 69
LogoutFilter, 67
movie creation, 63
RequestCacheAwareFilter, 67
RequestMapping annotation, 62
SecurityContextPersistenceFilter, 66
Servlet listener, 65
servlet-name value, 61
SessionManagementFilter, 68
terrormovies-servlet.xml, 64
URL access, 65
WEB-INF/terrormovies-servlet.xml file, 61
web.xml snippets, 61

X, Y, Z

X.509 authentication
applicationContext-security.xml file, 179
certificate generation, 180
certificate pkcs, 182
client certificate, 182
pom.xml file, 178
private key, 180
truststore, 181
workflow, 183
XML
AspectJ Maven Dependency, 132
AspectJ pointcut expressions, 131
moviesService Bean, 133
MoviesServiceImpl class, 131–132
XML namespace, 37
AUTHENTICATION_MANAGER, 39
AUTHENTICATION_PROVIDER, 39
<bean>–based configuration, 35
BeanDefinitionParser objects, 37
DEBUG, 39
Domain Specific Language (DSL), 35
FILTER_CHAIN, 39
FILTER_INVOCATION_DEFINITION_SOURCE, 39

XML namespace (*cont.*)
FILTER_SECURITY_METADATA_SOURCE, 39
GLOBAL_METHOD_SECURITY, 39
HTTP, 39
HTTP_FIREWALL, 39
integrated development environment (IDE), 35
JDBC_USER_SERVICE, 38
LDAP_PROVIDER, 38
LDAP_SERVER., 38
LDAP_USER_SERVICE, 38
load-up sequence, 36
META-INF directory, 36
METHOD_SECURITY_METADATA_SOURCE, 39
USER_SERVICE, 38

Pro Spring Security

Carlo Scarioni

Apress®

Pro Spring Security

Copyright © 2013 by Carlo Scarioni

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.

ISBN-13 (pbk): 978-1-4302-4818-7

ISBN-13 (electronic): 978-1-4302-4819-4

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

President and Publisher: Paul Manning
Lead Editor: Steve Anglin
Technical Reviewer: Manuel Jordan
Technical Reviewer: Arup Nanda
Editorial Board: Steve Anglin, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Kevin Shea
Copy Editor: Roger LeBlanc
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

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

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

To my wife, Monica, for her constant support and love. To my parents for always being there. And to my aunt Marisol, who we all miss so much.

—Carlo Scarioni

Download from Wow! eBook <www.wowebook.com>

Contents

About the Author xiii

About the Technical Reviewer xv

Acknowledgments xvii

Introduction xix

Chapter 1: The Scope of Security 1

The Network Security Layer 1

The Operating System Layer 1

The Application Layer 2

Authentication 2

Authorization 3

ACLs 4

Authentication and Authorization: General Concepts 4

What to Secure 7

More Security Concerns 7

Java Options for Security 8

Summary 8

Chapter 2: Introducing Spring Security 9

What Is Spring Security? 9

Where Does Spring Security Fit In? 10

Spring Security and Spring 12

Spring Framework: A Quick Overview 12
Dependency Injection 13
Aspect Oriented Programming (AOP) 14
An Initial Spring Security–Secured Application 15
Adding Spring Security (and Spring Core Itself) to the Project 18
Configuring the Web Project To Be Aware of Spring Security 21
Understanding the Simple Application 24
Summary 26
Chapter 3: Spring Security Architecture and Design 27
What Components Make Up Spring Security? 27
The 10,000-Foot View 27
The 1,000-Foot View 28
The 100-Foot View 29
Good Design and Patterns in Spring Security 55
Strategy Pattern 55
Decorator Pattern 55
SRP 56
DI 56
Summary 56
Chapter 4: Web Security 57
Introducing the Simple Example Application 57
The Special URLs 72
Custom Login Form 73
Basic HTTP Authentication 77
Digest Authentication 78
Remember-Me Authentication 80
Allowing Remember-Me Access to Selected Parts of the Application 81
Logging Out 83
The Session (javax.servlet.http.HttpSession) and the SecurityContext 84

Beyond Simple User Roles: Using Spring Expression Language to Secure the Web Layer 89
Extend with Your Own Expressions 91
Switching to a Different User 95
Session Management 98
Forcing the Request to HTTPS 102
Role Hierarchies 108
Summary 110

Chapter 5: Securing the Service Layer 111

The Limitations of Web-Level Security 111
What Is Business Service-Level Security? 111
Setting Up the Example for the Chapter 112
How the Described Actions Happen Under the Hood 117
Creating a Business Layer in Your Application 118
@RolesAllowed Annotation 120
Securing the Application Using SpEL Expressions 121
Securing the Data Returned from a Method 123
Filtering Collections Sent and Returned from Methods 125
Security Defined in XML 131
Security Without a Web Layer 133
Using AspectJ AOP instead of Spring AOP 141
Summary 151

Chapter 6: Configuring Alternative Authentication Providers 153

Database-Provided Authentication 153
Creating the Basic Tables 159
Using Groups 161
Using Existing Schemas 162
LDAP Authentication 163
Installing and Configuring LDAP 164
Other Attributes and Elements in the LDAP Spring Security Namespace 171

Authenticating with OpenID....172
Setting Up OpenID Authentication....173
OpenID Authentication Flow....175
Spring Security OpenID Namespace....176
X.509 Authentication....178
JAAS Authentication....184
Central Authentication Service (CAS) Authentication....190
Integrating CAS with a Different Authentication Provider....202
Summary....203
Chapter 7: Business Object Security with ACLs....205
The Security Example Application....205
Accessing Secured Objects....222
Filtering Returned Objects....226
Test Scenario 7-1....230
Securing the View Layer with ACLs....233
The Cost of ACLs....234
Summary....235
Chapter 8: Customizing and Extending Spring Security....237
Spring Security Extension Points....237
Plug into the Spring Security Event System....237
Authorization-Related Events....239
Authentication-Related Events....240
Session-Related Events....242
Your Own AuthenticationProvider and UserDetailsService....243
Password Encryption....256
New Voters in AccessDecisionManager....257
Nonvoter AccessDecisionManager Implementations....259
New Expression Root and SpEL....262
Non-JDBC AclService....262

Custom Security Filter 262
Handling Errors and Entry Points 265
Changing the Security Interceptor 269
Spring Security Extensions Project 271
Summary 272

Chapter 9: Integrating Spring Security with Other Frameworks and Languages 273

Spring Security with Struts 2 273
Spring Security with Spring Web Flow 280
SpEL-Based Security with Spring Web Flow 289
Spring Security in Other JVM Languages 291
Spring Security and Ruby (JRuby) 292
Web-Layer Security in Rails 293
Spring Security, Groovy, and Grails 297
Using Grails to Secure the Web Layer with URL Rules 297
Using Grails Security at the Method Level 300
Spring Security and Scala 301
Summary 309

Index 311

About the Author



Carlo Scarioni is a software engineer with a Computer Science degree from Universidad Central de Venezuela. He has worked in many industry fields in three different countries and has over nine years' experience. Currently living and working in London, Carlo focuses mostly on Java and Ruby software development. He has obtained multiple certifications in the Java space, including SCJP, SCBCD, and SpringSource Certified Spring Professional for Spring 2.5. Carlo fell in love with The Spring Framework when he first used it more than five years ago, and now he can't conceive starting a pure Java project without the aid of Spring and its related technologies. He is very passionate about his work and is constantly searching for the best ways to solve problems and trying to learn new and better technologies. He also likes to write on his blog as a way to give back something to the software community that he so much respects, and he is a DZone MVB (Most Valuable Blogger). You can follow Carlo's blog at `http://cscarioni.blogspot.com`.

About the Technical Reviewer



Manuel Jordan Elera is a self-taught developer and researcher who enjoys learning new technologies for his own experiments and creating new integrations.

Manuel won the 2010 Springy Award – Community Champion. In his scarce free time, he reads the Bible and composes music on his guitar. Manuel is a Senior Member in the Spring Community Forums, where he is known as *dr_pompeii*.

Manuel was Technical Reviewer for these books (all published by Apress):

- Pro SpringSource dm Server (2009)
- Spring Enterprise Recipes (2009)
- Spring Recipes (Second Edition) (2010)
- Pro Spring Integration (2011)
- Pro Spring Batch (2011)
- Pro Spring 3 (2012)
- Pro Spring MVC: With Web Flow (2012)

Read and contact him through his blog at `http://manueljordan.wordpress.com/`, and follow him on his Twitter account, `@dr_pompeii`.

Acknowledgments

This book is definitely the work of more than one person. The people involved in the preparation of this book have brought so much experience and quality to the final version that the end product is many times better than if I had done all the work myself. Their input ranged from improving text style, to introducing better ways to present concepts, to performing code reviews and suggesting general improvements that have made this book a much better reading experience.

I am talking, of course, about the great people at Apress who have been with me along the full journey of writing this book. I'm talking about Steve Anglin, who initiated me into the project, kept an eye from afar on the progress of the book, and tried to make sure I kept on track as much as possible. I'm talking about Kevin Shea, who was my main editorial contact and made sure that I stayed on schedule with the book and helped with advice and support. I'm talking about Tom Welsh, who had the great responsibility of reading every chapter as I was writing them and gave great input on each section, including helping with my use of English grammar as well as ways to make the different parts more attractive to potential readers. I am talking about Manuel Jordan, who not only read every single chapter in a very detailed way, but also took on the laborious job of evaluating and executing every single line of code and made sure that the book provides code samples that can be reproduced by the readers in their own environments. His input is greatly appreciated, and it is the difference between having a full book or a half a book. There were, of course, many more people in Apress involved in the full review phases of the book, and I want to say "thank you for your help" to all of them.

I would like to also thank the creators, committers and community of Spring and Spring Security for creating such an amazing piece of software and making it available to everyone. A big thank you to them for letting all developers share their knowledge and ways of work by freely distributing the source code of the different projects covered by the SpringSource umbrella. They make us all wiser and better developers.

Finally, I want to thank my wife for being with me all the time and motivating me to keep going forward.

—Carlo Scarioni