covers Spring 3.0

Spring IN ACTION

THIRD EDITION

Craig Walls

MANNING

Table of Contents

9 Securing Spring

This chapter covers

- Introducing Spring Security
- Securing web applications using servlet filters
- Authentication against databases and LDAP
- Transparently securing method invocations

Have you ever noticed that most people in television sitcoms don't lock their doors? It happens all the time. On *Seinfeld*, Kramer frequently let himself into Jerry's apartment to help himself to the goodies in Jerry's refrigerator. On *Friends*, the various characters often entered one another's apartments without warning or hesitation. Once, while in London, Ross even burst into Chandler's hotel room, narrowly missing Chandler in a compromising situation with Ross's sister.

In the days of *Leave it to Beaver*, it wasn't so unusual for people to leave their doors unlocked. But it seems crazy that in a day when we're concerned with privacy and security we see television characters enabling unhindered access to their apartments and homes.

It's a sad reality that there are villainous individuals roaming around seeking to steal our money, riches, cars, and other valuables. And it should be no surprise that as information is probably the most valuable item we have, crooks are looking for ways to steal our data and identity by sneaking into unsecured applications.

As software developers, we must take steps to protect the information that resides in our applications. Whether it's an email account protected with a username/password pair or a brokerage account protected with a trading PIN, security is a crucial *aspect* of most applications.

It's no accident that I chose to describe application security with the word "aspect." Security is a concern that transcends an application's functionality. For the most part, an application should play no part in securing itself. Although you could write security functionality directly into your application's code (and that's not uncommon), it's better to keep security concerns separate from application concerns.

If you're thinking that it's starting to sound as if security is accomplished using aspect-oriented techniques, you're right. In this chapter we're going to explore ways to secure your applications with aspects. But we won't have to develop those aspects ourselves—we're going to look at Spring Security, a security framework implemented with Spring AOP and servlet filters.[1]

9.1 Introducing Spring Security

Spring Security is a security framework that provides declarative security for your Spring-based applications. Spring Security provides a comprehensive security solution, handling authentication and authorization at both the web request level and at the method invocation level. Based on the Spring Framework, Spring Security takes full advantage of dependency injection (DI) and aspect-oriented techniques.

Spring Security got its start as Acegi Security. Acegi was a powerful security framework, but it had one big turn-off: it required a *lot* of XML configuration. I'll spare you the intricate details of what such a configuration may have looked like. Suffice it to say that it was common for a typical Acegi configuration to grow to several hundred lines of XML.

With version 2.0, Acegi Security became Spring Security. But the 2.0 release brought more than just a superficial name change. Spring Security 2.0 introduced a new security-specific XML namespace for configuring security in Spring. The new namespace, along with annotations and reasonable defaults, slimmed typical security configuration from hundreds of lines to only a dozen or so lines of XML. Spring Security 3.0, the most recent release, added SpEL to the mix, simplifying security configuration even more.

Spring Security tackles security from two angles. To secure web requests and restrict access at the URL level, Spring Security uses servlet filters. Spring Security can also secure method invocations using Spring AOP—proxying objects and applying advice that ensures that the user has proper authority to invoke secured methods.

[1] I'm probably going to get a lot of emails about this, but I have to say it anyway: servlet filters are a primitive form of AOP, with URL patterns as a kind of pointcut expression language. There… I've said it… I feel better now.

9.1.1 Getting started with Spring Security

No matter what kind of application you want to secure using Spring Security, the first thing to do is to add the Spring Security modules to the application's classpath. Spring Security 3.0 is divided into eight modules, as listed in table 9.1.

Table 9.1 Spring Security is partitioned into eight modules.

Module	Description
ACL	Provides support for domain object security through access control lists (ACLs)
CAS Client	Provides integration with JA-SIG's Central Authentication Service (CAS)
Configuration	Contains support for Spring Security's XML namespace
Core	Provides the essential Spring Security library
LDAP	Provides support for authentication using the Lightweight Directory Access Protocol (LDAP)
OpenID	Provides integration with the decentralized OpenID standard
Tag Library	Includes a set of JSP tags for view-level security
Web	Provides Spring Security's filter-based web security support

At the least, you'll want to include the Core and Configuration modules in your application's classpath. Spring Security is often used to secure web applications. That's certainly the case with the Spitter application, so we'll also need to add the web module. We'll also be taking advantage of Spring Security's JSP tag library, so we'll need to add that module to the mix.

Now we're ready to start declaring security configuration in Spring Security. Let's see how to get started with Spring Security's XML configuration namespace.

9.1.2 Using the Spring Security configuration namespace

When Spring Security was known as Acegi Security, all of the security elements were configured as `<bean>`s in the Spring application context. A common Acegi configuration scenario would contain dozens of `<bean>` declarations and span multiple pages. The long and short of it was that Acegi configuration was often longer than it was short.

Spring Security comes with a security-specific namespace that greatly simplifies security configuration in Spring. This new namespace, along with some sensible default behavior, reduces a typical security configuration from over 100 lines of XML to a dozen or so.

The only thing to do in preparation for using the security namespace is to include it in the XML file by adding the namespace declaration:

Listing 9.1 Adding the Spring Security namespace to a Spring configuration XML file

```
<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-3.0.xsd">

</beans>
```

security:-prefixed elements go here

For the Spitter application, we've separated all of the security-specific configuration into a separate Spring configuration file called spitter-security.xml. Since all of the configuration in this file will be from the security namespace, we've changed the security namespace to be the primary namespace for that file.

Listing 9.2 Using the security namespace as the default namespace

```
<beans:beans xmlns:beans="http://www.springframework.org/schema/beans"
    xmlns="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-3.0.xsd">

</beans:beans>
```

Non-prefixed security elements go here

With the security namespace as the primary namespace, we can avoid adding those pesky `security:` prefixes on all of the elements.

All of the Spring Security pieces are falling into place nicely. Now we're ready to add web-level security to the Spitter application.

9.2 Securing web requests

Everything you do with a Java web application starts with an `HttpServletRequest`. And if the request is the access point to a web application, then that's where security for a web application should begin.

The most basic form of request-level security involves declaring one or more URL patterns as requiring some level of granted authority and preventing users without that authority from accessing the content behind those URLs. Taking it a step further, you may want to require that certain URLs can only be accessed over HTTPS.

Before you can restrict access to users with certain privileges, there must be a way to know who's using the application. Therefore, the application will need to authenticate the user, prompting them to log in and identify themselves.

Spring Security supports these and many other forms of request-level security. To get started with web security in Spring, we must set up the servlet filters that provide the various security features.

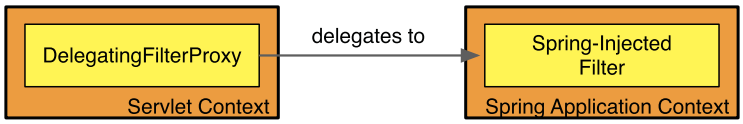

Figure 9.1 `DelegatingFilterProxy` proxies filter handling to a delegate filter bean in the Spring application context.

9.2.1 Proxying servlet filters

Spring Security employs several servlet filters to provide various aspects of security. As you might imagine, this could mean several `<filter>` declarations in your application's web.xml file. But rest easy—thanks to a little Spring magic, we'll only need to configure one filter in the application's web.xml file. Specifically, we'll need to add the following `<filter>`:

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

`DelegatingFilterProxy` is a special servlet filter that, by itself, doesn't do much. Instead, it delegates to an implementation of `javax.servlet.Filter` that's registered as a `<bean>` in the Spring application context, as illustrated in figure 9.1.

In order to do their job, Spring Security's filters must be injected with some other beans. It's not possible to inject beans into servlet filters registered in web.xml. But by using `DelegatingFilterProxy`, we can configure the actual filter in Spring, taking full advantage of Spring's support for dependency injection.

The value given as `DelegatingFilterProxy`'s `<filter-name>` is significant. This is the name used to look up the filter bean from the Spring application context. Spring Security will automatically create a filter bean whose ID is `springSecurityFilterChain`, so that's the name we've given to `DelegatingFilterProxy` in web.xml.

As for the `springSecurityFilterChain` bean itself, it's another special filter known as `FilterChainProxy`. It's a single filter that chains together one or more additional filters. Spring Security relies on several servlet filters to provide different security features. But you should almost never need to know these details, as you likely won't need to explicitly declare the `springSecurityFilterChain` bean or any of the filters it chains together. Spring Security will automatically create those beans for us when we configure the `<http>` element, which we'll do next.

9.2.2 Configuring minimal web security

Early versions of Spring Security required a seemingly endless amount of XML configuration to set up basic security features. In contrast, using recent versions of Spring Security, the following snippet of XML packs a lot of punch:

```
<http auto-config="true">
  <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

These humble three lines of XML configure Spring security to intercept requests for all URLs (as specified by the Ant-style path in the `pattern` attribute of `<intercept-url>`) and restrict access to only authenticated users who have the `ROLE_SPITTER` role. The `<http>` element automatically sets up a `FilterChainProxy` (which is delegated to by the `DelegatingFilterProxy` we configured in web.xml) and all of the filter beans in the chain.

In addition to those filter beans, we also get a few more freebies by setting the `auto-config` attribute to `true`. Autoconfiguration gives our application a free login page, support for HTTP Basic authentication, and support for logging out. In fact, setting `auto-config` to `true` is equivalent to explicitly asking for those features like this:

```
<http>
    <form-login />
    <http-basic />
    <logout />
  <intercept-url pattern="/**" access="ROLE_SPITTER" />
</http>
```

Let's dig deeper into what these features give us and see how to use them.

LOGGING IN VIA A FORM

One of the benefits of setting `auto-config` to `true` is that Spring Security will automatically generate a login page for you. Here's the HTML for that form.

Listing 9.3 Spring Security can automatically generate a simple login form for you.

```
<html>
  <head><title>Login Page</title></head>
  <body onload='document.f.j_username.focus();'>
    <h3>Login with Username and Password</h3>
    <form name='f' method='POST'
          action='/Spitter/j_spring_security_check'>
      <table>
        <tr><td>User:</td><td>
            <input type='text' name='j_username' value=''>
          </td></tr>
        <tr><td>Password:</td><td>
            <input type='password' name='j_password'/>
          </td></tr>
        <tr><td colspan='2'><input name="submit" type="submit"/></td></tr>
        <tr><td colspan='2'><input name="reset" type="reset"/></td></tr>
      </table>
    </form>
  </body>
</html>
```

Authentication filter path

Username field

Password field

You can get to the automatically generated login form via the path `/spring_security_login` relative to the application's context URL. For example,

when accessing the Spitter application on localhost, that URL is http://localhost:8080/Spitter/spring_security_login.

At first it may seem like a great deal that Spring Security gives you a login form for free. But as you can see, the form is simple and not a lot should be said about its aesthetics. It's plain and we'll probably want to replace it with a login page of our own design.

To put our own login page in place, we'll need to configure a `<form-login>` element to override the default behavior:

```
<http auto-config="true" use-expressions="false">
  <form-login login-processing-url="/static/j_spring_security_check"
              login-page="/login"
              authentication-failure-url="/login?login_error=t"/>
</http>
```

The `login` attribute specifies a new context-relative URL for the login page. In this case we state that the login page will reside at `/login` which is ultimately handled by a Spring MVC controller. Likewise, if authentication fails, the `authentication-failure-url` attribute is set to send the user back to the same login page.

Note that we've set the `login-processing-url` to `/static/j_spring_security_check`. This is the URL that the login form will submit back to to authenticate the user.

Even though we may not want to keep the user-generated login form, we can learn a lot from it. For starters, we know that Spring Security will process the login request at the path `/Spitter/j_spring_security_check`. And it's clear that the username and password should be submitted in the request as fields named `j_username` and `j_password`. Armed with that information, we can create our own custom login page.

For the Spitter application the new login page is a JSP that's served up by a Spring MVC controller. The JSP itself is shown next.

Listing 9.4 The Spitter application uses a custom login page defined as JSP.

```
<%@ taglib prefix="s" uri="http://www.springframework.org/tags"%>
<div>
   <h2>Sign in to Spitter</h2>

   <p>
    If you've been using Spitter from your phone,
    then that's amazing...we don't support IM yet.
   </p>

   <spring:url var="authUrl"
          value="/static/j_spring_security_check" />
   <form method="post" class="signin" action="${authUrl}">

    <fieldset>
    <table cellspacing="0">
    <tr>
    <th><label for="username_or_email">Username or Email</label></th>
    <td><input id="username_or_email"
```

Authentication filter path

```
            name="j_username"
            type="text" />                    <-- Username field
      </td>
    </tr>
    <tr>
    <th><label for="password">Password</label></th>
      <td><input id="password"
                 name="j_password"                                  Password
                 type="password" />                                 field
          <small><a href="/account/resend_password">Forgot?</a></small>
      </td>
    </tr>
    <tr>
    <th></th>
    <td><input id="remember_me"
        name="_spring_security_remember_me"
        type="checkbox"/>                     <-- Remember-me box
        <label for="remember_me"
               class="inline">Remember me</label></td>
    </tr>
    <tr>
    <th></th>
    <td><input name="commit" type="submit" value="Sign In" /></td>
    </tr>
    </table>
    </fieldset>
   </form>

   <script type="text/javascript">
    document.getElementById('username_or_email').focus();
   </script>
</div>
```

Although our login page is different than the one that Spring Security gives us out of the box, the key thing is that the form submits `j_username` and `j_password` parameters with the user's credentials. Everything else is decoration.

Also note that listing 9.4 includes a "remember me" checkbox. We'll discuss the details of how that works later in section 9.4.4. But for now, let's see Spring Security's support for HTTP Basic authentication.

HANDLING BASIC AUTHENTICATION

Form-based authentication is ideal for human users of an application. But in chapter 11, we'll see how to turn some of our web application's pages into a RESTful API. When the user of the application is another application, prompting for login with a form just won't do.

HTTP Basic authentication is a way to authenticate a user to an application directly in the HTTP request itself. You may have seen HTTP Basic authentication before. When encountered by a web browser, it prompts the user with a plain modal dialog box.

But that's just how it's manifested in a web browser. In reality, it's an HTTP 401 response, indicating that a username and password must be presented with the

request. This makes it suitable as a means for REST clients to authenticate against the services that they're consuming.

Not much customization is available with `<http-basic>`. HTTP Basic authentication is either turned on or it's not. So rather than dwell on the topic any further, let's move on to see what the `<logout>` element gives us.

LOGGING OUT

The `<logout>` element sets up a Spring Security filter that will invalidate a user session. When used as is, the filter set up by `<logout>` is mapped to /j_spring_security_logout. But so that this doesn't collide with how we've set up `DispatcherServlet`, we need to override the filter's URL much as we did for the login form. To do that, we need to set the `logout-url` attribute:

```
<logout logout-url="/static/j_spring_security_logout"/>
```

That wraps up our discussion of what autoconfiguration gives us. But there's more to explore in Spring Security. Let's take a closer look at the `<intercept-url>` element and see how it controls access at the request level.

9.2.3 *Intercepting requests*

In the previous section, we saw a simple example of the `<intercept-url>` element. But we didn't dig into it much… until now.

The `<intercept-url>` element is the first line of defense in the request-level security game. Its `pattern` attribute is given a URL pattern that will be matched against incoming requests. If any requests match the pattern, then that `<intercept-url>`'s security rules will be applied.

Let's revisit the `<intercept-url>` element from before:

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

The `pattern` attribute takes an Ant-style path by default. But if you'd prefer, setting the `<http>` element's `path-type` attribute to `regex` will change the `pattern` to taking regular expressions.

In this case, we've set the `pattern` attribute to `/**`, indicating that we want all requests, regardless of the URL, to require ROLE_SPITTER access. The `/**` has a broad reach, but you can be more specific.

Suppose that some special areas of the Spitter application are restricted to administrative users. For that, we can insert the following `<intercept-url>` just before the one we already have:

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

Where our first `<intercept-url>` entry makes sure that the user has ROLE_SPITTER authority for most of the application, this `<intercept-url>` restricts access to the /admin branch of the site's hierarchy to users with ROLE_ADMIN authority.

You can use as many `<intercept-url>` entries as you like to secure various paths in your web application. But it's important to know that the `<intercept-url>` rules are

applied top to bottom. Therefore, this new `<intercept-url>` should be placed before the original one or else it'll be eclipsed by the broad scope of the `/**` path.

SECURING WITH SPRING EXPRESSIONS

Listing required authorities is simple enough, but it's somewhat one-dimensional. What if you wanted to express security constraints that are based on more than just granted privileges?

In chapter 2, we saw how to use the Spring Expression Language (SpEL) as an advanced technique for wiring bean properties. As of version 3.0, Spring Security also supports SpEL as a means for declaring access requirements. To enable it, we must set the `use-expressions` attribute of `<http>` to `true`:

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

Now we can start using SpEL expressions in the `access` attribute. Here's how to use a SpEL expression to require ROLE_ADMIN access for the `/admin/**` URL pattern:

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

This `<intercept-url>` is effectively the same as the one we started with, except that it uses SpEL. The `hasRole()` expression evaluates to `true` if the current user has been granted the given authority. But `hasRole()` is only one of several security-specific expressions supported. Table 9.2 lists all of the SpEL expressions added by Spring Security 3.0.

Table 9.2 Spring Security extends the Spring Expression Language with a several security-specific expressions.

Security expression	What it evaluates to
`authentication`	The user's authentication object
`denyAll`	Always evaluates to `false`
`hasAnyRole(list of roles)`	`true` if the user has been granted any of the roles specified
`hasRole(role)`	`true` if the user has been granted the specified role
`hasIpAddress(IP Address)`	The user's IP address (only available in web security)
`isAnonymous()`	`true` if the current user is an anonymous user
`isAuthenticated()`	`true` if the current user is not anonymous
`isFullyAuthenticated()`	`true` if the current user is neither an anonymous nor a remember-me user
`isRememberMe()`	`true` if the current user was automatically authenticated via remember-me
`permitAll`	Always evaluates to `true`
`principal`	The user's principal object

With Spring Security's SpEL expressions at our disposal, we can do more than just limit access based on a user's granted authorities. For example, if you wanted to lock down the `/admin/**` URLs to not only require ROLE_ADMIN, but to also only be allowed from a given IP address, you might declare an `<intercept-url>` like this:

```
<intercept-url pattern="/admin/**"
    access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.2')"/>
```

With SpEL-based security constraints, the possibilities are virtually endless. I'll bet that you're already dreaming up interesting security constraints based on SpEL.

But for now, let's look at another one of `<intercept-url>`'s tricks: enforcing channel security.

FORCING REQUESTS TO HTTPS

Submitting data across HTTP can be a risky proposition. It may not be a big deal to send a spittle message in the clear over HTTP. But if you're passing sensitive information such as passwords and credit card numbers across HTTP, then you're asking for trouble. That's why sensitive information should be sent encrypted over HTTPS.

Working with HTTPS seems simple enough. All you have to do is add an `s` after the `http` in a URL and you're set. Right?

That's true, but it places responsibility for using the HTTPS channel in the wrong place. If you have dozens or hundreds of links and form actions that should be going to an HTTPS URL, it's too easy to forget to add that S. Chances are good that you'll miss one or two of them. Or you may overcorrect and use HTTPS in places where it's unnecessary.

The `<intercept-url>` element's `requires-channel` attribute shifts the responsibility for channel enforcement into the Spring Security configuration.

As an example, consider the Spitter application's registration form. Although Spitter doesn't ask for credit card numbers or social security numbers or anything terribly sensitive, the users may want that information to be kept private. In that case, we should configure an `<intercept-url>` element for the /spitter/form like this:

```
<intercept-url pattern="/spitter/form" requires-channel="https"/>
```

Anytime a request comes in for /spitter/form, Spring Security will see that it requires the `https` channel and automatically redirect the request to go over HTTPS. Likewise, the home page doesn't require HTTPS, so we can declare that it always should be sent over HTTP:

```
<intercept-url pattern="/home" requires-channel="http"/>
```

So far we've seen how to secure web applications as requests are made. The assumption has been that security would involve stopping a user from accessing a URL that they're not authorized to use. But it's also a good idea to never show links that a user won't be able to follow. Let's see how Spring Security offers security in the view.

9.3 Securing view-level elements

To support security in the view layer, Spring Security comes with a JSP tag library.[2] This tag library is small and includes only three tags, as listed in table 9.3.

Table 9.3 Spring Security supports security in the view layer with a JSP tag library.

JSP tag	What it does
`<security:accesscontrollist>`	Allows the body of the tag to be rendered if the currently authenticated user has one of the stipulated permissions in the specified domain object
`<security:authentication>`	Accesses properties of the current user's authentication object
`<security:authorize>`	Allows the body of the tag to be rendered if a specified security constraint has been met

To use the JSP tag library, we'll need to declare it in the JSP files where it's used:

```
<%@ taglib prefix="security"
           uri="http://www.springframework.org/security/tags" %>
```

Once the tag library has been declared in the JSP file, we're ready to use it. Let's look at each of the three JSP tags that come with Spring Security and see how they work.

9.3.1 Accessing authentication details

One of the simplest things that the Spring Security JSP tag library can do for us is provide convenient access to the user's authentication information. For example, it's common for websites to display a "welcome" or "hello" message in the page header, identifying the user by their username. That's precisely the kind of thing that the `<security:authentication>` can do for us. For example:

```
Hello <security:authentication property="principal.username" />!
```

The `property` attribute identifies a property of the user's authentication object. The properties available will vary depending on how the user was authenticated. But you can count on a few common properties to be available, including those listed in table 9.4.

In our example, the property being rendered is actually the nested `username` property of the `principal` property.

When used as shown in the previous example, `<security:authentication>` will render the property's value in the view. But if you'd rather assign it to a variable, then simply specify the name of the variable in the `var` attribute:

```
<security:authentication property="principal.username"
        var="loginId"/>
```

[2] If you prefer Velocity over JSP for rendering views, Spring Security also comes with a set of Velocity macros that are similar to its JSP tags.

Table 9.4 You can access several of the user's authentication details using the `<security:authentication>` JSP tag.

Authentication property	Description
`authorities`	A collection of `GrantedAuthority` objects that represent the privileges granted to the user
`credentials`	The credentials that were used to verify the principal (commonly, this is the user's password)
`details`	Additional information about the authentication (IP address, certificate serial number, session ID, and so on)
`principal`	The user's principal

The variable is created in page scope by default. But if you'd rather create it in some other scope such as request or session (or any of the scopes available from `javax.servlet.jsp.PageContext`), you can specify it via the `scope` attribute. For example, to create the variable in request scope, use the `<security:authentication>` tag like this:

```
<security:authentication property="principal.username"
        var="loginId" scope="request" />
```

The `<security:authentication>` tag is useful, but it's just the start of what Spring Security's JSP tag library can do. Let's see how to conditionally render content depending on the user's privileges.

9.3.2 Rendering with authorities

Sometimes portions of the view should or shouldn't be rendered, depending on what the user is privileged to do. There's no point in showing a login form to a user who's already logged in or in showing a personalized greeting to a user who's not logged in.

Spring Security's `<security:authorize>` JSP tag conditionally renders a portion of the view depending on the user's granted authorities. For example, in the Spitter application we don't want to show the form for adding a new spittle unless the user has the ROLE_SPITTER role. The following listing shows how to use the `<security:authorize>` tag to display the spittle form if the user has ROLE_SPITTER authority.

Listing 9.5 Conditional rendering with the `<security:authorize>` tag

```
<sec:authorize access="hasRole('ROLE_SPITTER')">

    <s:url value="/spittles" var="spittle_url" />
      <sf:form modelAttribute="spittle"

                action="${spittle_url}">
        <sf:label path="text"><s:message code="label.spittle"
                        text="Enter spittle:"/></sf:label>
        <sf:textarea path="text" rows="2" cols="40" />
            <sf:errors path="text" />
```

Only with ROLE_SPITTER authority

```
        <br/>
        <div class="spitItSubmitIt">
          <input type="submit" value="Spit it!"
               class="status-btn round-btn disabled" />
        </div>
      </sf:form>
    </sec:authorize>
```

The `access` attribute is given a SpEL expression whose result determines whether `<security:authorize>`'s body is rendered. Here we're using the `hasRole ('ROLE_SPITTER')` expression to ensure that the user has the ROLE_SPITTER role. But you have the full power of SpEL at your disposal when setting the `access` attribute, including the Spring Security-provided expressions listed in table 9.2.

With these expressions available, you can cook up some interesting security constraints. For example, imagine that the application has some administrative functions that are only available to the user whose username is habuma. Maybe you'd use the `isAuthenticated()` and `principal` expressions like this:

```
<security:authorize
   access="isAuthenticated() and principal.username=='habuma'">
  <a href="/admin">Administration</a>
</security:authorize>
```

I'm sure you can dream up even more interesting expressions than that. I'll leave it up to your imagination to concoct more security constraints. The options are virtually limitless with SpEL.

But one thing about the example that I dreamt up still bugs me. Though I might want to restrict the administrative functions to habuma, perhaps doing it with a SpEL expression isn't ideal. Sure, it'll keep the link from being rendered in the view. But nothing's stopping anyone from manually entering the /admin URL in the browser's address line.

Drawing on what we learned earlier in this chapter, that should be an easy thing to fix. Adding a new `<intercept-url>` in the security configuration will tighten security around the /admin URL:

```
<intercept-url pattern="/admin/**"
    access="hasRole('ROLE_ADMIN') and hasIpAddress('192.168.1.2')"/>
```

Now the admin functionality is locked down. The URL is secured and the link to the URL won't appear unless the user is authorized to use it. But to do that we had to declare the SpEL expression in two places—in `<intercept-url>` and in the `<security:authorize>` tag's `access` attribute. Wouldn't it make more sense to only show the URL if the URL's security constraint was met?

That's what the `<security:authorize>` tag's `url` attribute is for. Unlike the `access` attribute where the security constraint is explicitly declared, the `url` attribute indirectly refers to the security constraints for a given URL pattern. Since we've already declared security constraints for /admin in the Spring Security configuration, we can use the `url` attribute like this:

```
<security:authorize url="/admin/**">
  <spring:url value="/admin" var="admin_url" />
  <br/><a href="${admin_url}">Admin</a>
</security:authorize>
```

Since the /admin URL is restricted to only authenticated users who have ROLE_ADMIN authority and to requests coming from a specific IP address, the body of the `<security:authorize>` tag will only be rendered if those conditions are met.

> **What about `<security:authorize>`'s other attributes?**
>
> In addition to the `access` and `url` attributes, `<security:authorize>` has three other attributes: `ifAllGranted`, `ifAnyGranted`, and `ifNotGranted`. These attributes make `<security:authorize>` conditionally render depending on what authorities have or haven't been granted to the user.
>
> Prior to Spring Security 3.0, these were the only attributes available for `<security:authorize>`. But with the introduction of SpEL and the `access` attribute, they become obsolete. They're still available, but the `access` attribute can do the same things and much more.

We've now seen how to declare various forms of security at the web layer. One question remains: where is user information kept? In other words, when someone tries to log in to the application, what repository of user information does Spring Security use to authenticate against?

Put simply, Spring Security is flexible enough to authenticate against virtually any kind of user repository. Let's look at a few of the authentication options Spring Security offers.

9.4 Authenticating users

Every application's a little different. That truth is evident in how every application stores user information. Sometimes it's kept in a relational database. Other times it might be in an LDAP-enabled directory. Some applications rely on a decentralized user identity system. And some may employ more than one strategy.

Fortunately, Spring Security is flexible and can handle almost any authentication strategy you need. Spring Security is prepared to cover many common authentication scenarios, including authenticating users against

- In-memory (Spring-configured) user repositories
- JDBC-based user repositories
- LDAP-based user repositories
- OpenID decentralized user identity systems
- Central Authentication System (CAS)
- X.509 certificates
- JAAS-based providers

If none of the out-of-the-box options suit you, then you can easily implement your own authentication strategy and wire it in.

Let's have a deeper look at a few of the most commonly used authentication options that Spring Security offers.

9.4.1 Configuring an in-memory user repository

One of the easiest authentication options available is to declare the user details directly in the Spring configuration. This is done by creating a user service using the `<user-service>` element from Spring Security's XML namespace:

```
<user-service id="userService">
  <user name="habuma" password="letmein"
                authorities="ROLE_SPITTER,ROLE_ADMIN"/>
  <user name="twoqubed" password="longhorns"
                authorities="ROLE_SPITTER"/>
  <user name="admin" password="admin"
                authorities="ROLE_ADMIN"/>
</user-service>
```

A user service is effectively a data access object that looks up user details when given a user's login ID. In the case of `<user-service>`, those user details are declared within `<user-service>`. There's one `<user>` element for each user that can log in to the application. The `name` and `password` attributes respectively specify the login name and password. Meanwhile, the `authorities` attribute is set to a comma-separated list of authorities—the things that the user is allowed to do.

Recall that earlier (in section 9.2.3) we configured Spring Security to restrict access to all URLs to only users with ROLE_SPITTER authority. In this case, the habuma and twoqubed users would be granted access, but the admin user would be denied.

The user service is now ready and waiting to look up user details for authentication. All that's left is to wire it into Spring Security's authentication manager:

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

The `<authentication-manager>` element registers an authentication manager. More specifically, it registers an instance of `ProviderManager`, an authentication manager that delegates authentication responsibility to one or more authentication providers. In this case, it's an authentication provider that relies on a user service to provide user details. We happen to have a user service handy. So all we have to do is wire it in through the `user-service-ref` attribute of `<authentication-provider>`.

Here we've declared the authentication provider and the user service independently and wired them together. Optionally, if it suits you better, you could also embed the user service within the authentication provider:

```
<authentication-provider>
  <user-service id="userService">
    <user name="habuma" password="letmein"
```

```
            authorities="ROLE_SPITTER,ROLE_ADMIN"/>
    ...
  </user-service>
</authentication-provider>
```

There's no significant benefit in embedding the `<user-service>` within `<authentication-provider>`, but if it helps you organize your Spring XML configuration, then that option is available.

Defining user details in the Spring application context is convenient for testing or when you're first starting to add security to your application. But it's not a very realistic way of managing users in a production application. More often user details are kept in a database or a directory server. Let's see how to register a user service that looks for user details in a relational database.

9.4.2 Authenticating against a database

Many applications store user information, including the username and password, in a relational database. If that's how your application keeps user information, Spring Security's `<jdbc-user-service>` is a good choice for your application.

The `<jdbc-user-service>` is used the same way that `<user-service>` is used. This includes wiring it into `<authentication-provider>`'s `user-service-ref` attribute or embedding it within `<authentication-provider>`. Here we're configuring a basic `<jdbc-user-service>` with an `id` so that it can be declared independently and wired into the `<authentication-provider>`:

```
<jdbc-user-service id="userService"
    data-source-ref="dataSource" />
```

The `<jdbc-user-service>` element uses a JDBC data source—wired in through its `data-source-ref` attribute—to query a database for user details. Without any further configuration, the user service queries for user information using the following SQL:

```
select username,password,enabled
  from users
 where username = ?
```

And, although we're talking about user authentication right now, part of the authentication involves looking up the user's granted authorities. By default, the basic `<jdbc-user-service>` configuration will use the following SQL to look up authorities given a username:

```
select username,authority
  from authorities
 where username = ?
```

This is great if your application's database happens to store user details and authorities in tables that match those queries. But I'll bet that's not the case for most applications. In fact, in the case of the Spitter application, user details are kept in the `spitter` table. Clearly the default behavior isn't going to work.

Table 9.5 The attributes of `<jdbc-user-service>` that can change the SQL used to query for user details

Attribute	What it does
`users-by-username-query`	Queries for a user's username, password, and enabled status given the username
`authorities-by-username-query`	Queries for a user's granted authorities given the username
`group-authorities-by-username-query`	Queries for a user's group authorities given the username

Fortunately, `<jdbc-user-service>` can easily be configured to use whatever queries best fit your application. Table 9.5 describes the attributes that can be used to tweak `<jdbc-user-service>`'s behavior.

For the Spitter application, we'll set the `users-by-username-query` and `authorities-by-username-query` attributes as follows:

```
<jdbc-user-service id="userService"
       data-source-ref="dataSource"
       users-by-username-query=
          "select username, password, true from spitter where username=?"
       authorities-by-username-query=
          "select username,'ROLE_SPITTER' from spitter where username=?" />
```

In the Spitter application, the username and password are stored in the `spitter` table in the `username` and `password` properties, respectively. But we haven't really considered the idea of a user being enabled or disabled and have been assuming that all users are enabled. So we've written the SQL to always return `true` for all users.

We also haven't given much thought to giving Spitter users different levels of authority. All Spitter users have the same authorities. In fact, the Spitter database schema doesn't have a table for storing user authorities. Therefore, we've set `authorities-by-username-query` with a concocted query that gives all users ROLE_SPITTER authority.

While relational databases are commonly where an application's user details are kept, just as often (or perhaps more often) you'll find applications that need to authenticate against a directory server using LDAP. Let's see how to configure Spring Security to use LDAP as a user repository.

9.4.3 *Authenticating against LDAP*

We've all seen an organizational chart or two before. Most organizations are structured hierarchically. Employees report to supervisors, supervisors to directors, directors to vice presidents, and so forth. Within that hierarchy you'll often find a similarly hierarchical set of security rules. Human resources personnel are probably granted different privileges than accounting personnel. Supervisors probably have more open access than those that report to them.

As useful as relational databases can be, they don't do well representing hierarchical data. LDAP directories, on the other hand, excel at storing information hierarchically. For that reason, it's common to find a company's organizational structure represented in an LDAP directory. Alongside, you'll often find the company's security constraints mapped to the entries in the directory.

To use LDAP-based authentication, we'll first need to use Spring Security's LDAP module and configure LDAP authentication within the Spring application context. When it comes to configuring LDAP authentication, we have two choices:

- With an LDAP-oriented authentication provider
- With an LDAP-oriented user service

For the most part, it's an even choice on which you should use. But there are some small considerations to make when choosing one over the other.

DECLARING AN LDAP AUTHENTICATION PROVIDER

For the in-memory and JDBC-based user services, we declared an `<authentication-provider>` and wired in the user service. We can do the same thing for an LDAP-oriented user service (and I'll show you how in a moment). But a more direct way is to use a special LDAP-oriented authentication provider by declaring an `<ldap-authentication-provider>` within the `<authentication-manager>`:

```
<authentication-manager alias="authenticationManager">
      <ldap-authentication-provider
          user-search-filter="(uid={0})"
          group-search-filter="member={0}"/>
    </authentication-manager>
```

The `user-search-filter` and `group-search-filter` attributes are used to provide a filter for the base LDAP queries, which are used to search for users and groups. By default, the base queries for both users and groups are empty, indicating that the search will be done from the root of the LDAP hierarchy. But we can change that by specifying a query base:

```
<ldap-user-service id="userService"
             user-search-base="ou=people"
             user-search-filter="(uid={0})"
             group-search-base="ou=groups"
             group-search-filter="member={0}" />
```

The `user-search-base` attribute provides a base query for finding users. Likewise, the `group-search-base` specifies the base query for finding groups. Rather than search from the root, we've specified that users be searched for where the organization unit is `people`. And groups should be searched for where the organizational unit is `groups`.

CONFIGURING PASSWORD COMPARISON

The default strategy to authenticate against LDAP is to perform a bind operation, authenticating the user directly to the LDAP server. Another option is to perform a comparison operation. This involves sending the entered password to the LDAP directory and asking the server to compare the password against a user's password

attribute. Because the comparison is done within the LDAP server, the actual password remains secret.

If you'd rather authenticate by doing a password comparison, you can do so by declaring so with the `<password-compare>` element:

```
<ldap-authentication-provider
    user-search-filter="(uid={0})"
    group-search-filter="member={0}">
  <password-compare />
</ldap-authentication-provider>
```

As declared here, the password given in the login form will be compared with the value of the `userPassword` attribute in the user's LDAP entry. If the password is kept in a different attribute, then specify the password attribute's name with `password-attribute`:

```
<password-compare hash="md5"
    password-attribute="passcode" />
```

It's nice that the actual password is kept secret on the server when doing server-side password comparison. But the attempted password is still passed across the wire to the LDAP server and could be intercepted by a hacker. To prevent that, you can specify an encryption strategy by setting the `hash` attribute to one of the following values:

- `{sha}`
- `{ssha}`
- `md4`
- `md5`
- `plaintext`
- `sha`
- `sha-256`

In the example, we've encrypted passwords using MD5 by setting `hash` to `md5`.

REFERRING TO A REMOTE LDAP SERVER

The one thing I've left out up until now is where the LDAP server and data actually reside. We've happily been configuring Spring to authenticate against an LDAP server, but where's that server?

By default, Spring Security's LDAP authentication assumes that the LDAP server is listening on port 33389 on localhost. But if your LDAP server is on another machine, then you can use the `<ldap-server>` element to configure the location:

```
<ldap-server url="ldap://habuma.com:389/dc=habuma,dc=com" />
```

Here we use the `url` attribute to specify the location of the LDAP server.[3]

[3] Don't even try to use this LDAP URL. It's just an example. No LDAP server is actually listening there.

CONFIGURING AN EMBEDDED LDAP SERVER

If you don't happen to have an LDAP server laying around waiting to be authenticated against, then the `<ldap-server>` can also be used to configure an embedded LDAP server. Just leave off the `url` parameter. For example:

```
<ldap-server root="dc=habuma,dc=com" />
```

The `root` attribute is optional. But it defaults to `dc=springframework,dc=org`, which I suspect isn't what you'll want to use as the root for your LDAP server.

When the LDAP server starts, it will attempt to load data from any LDIF files that it can find in the classpath. LDIF (LDAP Data Interchange Format) is a standard way of representing LDAP data in a plain text file. Each record is comprised of one or more lines, each containing a name:value pair. Records are separated from each other by blank lines.[4]

If you'd rather be more explicit about which LDIF file gets loaded, you can use the `ldif` attribute:

```
<ldap-server root="dc=habuma,dc=com"
             ldif="classpath:users.ldif" />
```

Here we specifically ask the LDAP server to load its content from the users.ldif file at the root of the classpath. In case you're curious, the following listing shows the LDIF file that we've been using.

Listing 9.6 A sample LDIF file used to load user details into LDAP

```
dn: ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: groups

dn: ou=people,dc=habuma,dc=com
objectclass: top
objectclass: organizationalUnit
ou: people

dn: uid=habuma,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
cn: Craig Walls
sn: Walls
uid: habuma
userPassword: password

dn: uid=jsmith,ou=people,dc=habuma,dc=com
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
```

[4] See http://tools.ietf.org/html/rfc2849 for more details on the LDIF specification.

```
cn: John Smith
sn: Smith
uid: jsmith
userPassword: password

dn: cn=spitter,ou=groups,dc=habuma,dc=com
objectclass: top
objectclass: groupOfNames
cn: spitter
member: uid=habuma,ou=people,dc=habuma,dc=com
```

Whether your user authenticates against a database or an LDAP directory, it's always more convenient for them not to have to directly authenticate at all. Let's see how to configure Spring Security to remember a user so that they don't have to log in every time they visit an application.

9.4.4 Enabling remember-me functionality

It's important for an application to be able to authenticate users. But from the user's perspective, it'd be nice if the application didn't always prompt them with a login every time they use it. That's why many websites offer remember-me functionality so that you can log in once and then be remembered by the application when you come back to it later.

Spring Security makes it easy to add remember-me functionality to an application. To turn on remember-me support, all we need to do is add a `<remember-me>` element within the `<http>` element:

```
<http auto-config="true" use-expressions="true">
  ...
  <remember-me
      key="spitterKey"
      token-validity-seconds="2419200" />
</http>
```

Here, we've turned on remember-me functionality along with a bit of special configuration. If you use the `<remember-me>` element with no attributes, this feature is accomplished by storing a token in a cookie that's valid for up to two weeks. But here we've specified that the token should stay valid for up to four weeks (2,419,200 seconds).

The token that's stored in the cookie is made up of the username, password, an expiration date, and a private key—all encoded in an MD5 hash before being written to the cookie. By default, the private key is `SpringSecured`, but we've set it to `spitterKey` to make it specific to the Spitter application.

Simple enough. Now that remember-me functionality is enabled, we'll need to make a way for users to indicate that they'd like the application to remember them. For that, the login request will need to include a `_spring_security_remember_me` parameter. A simple checkbox in the login form ought to do the job:

```
<input id="remember_me" name="_spring_security_remember_me"
       type="checkbox"/>
<label for="remember_me" class="inline">Remember me</label>
```

Chapter 9. Securing Spring. Spring in Action, Third Edition, ISBN: 9781935182351

Up until now we've been mostly focused on securing web requests. Since Spring Security is often used to secure web applications, it tends to be forgotten that it can also be used to secure method invocations. Let's look at Spring Security's support for method-level security.

9.5 Securing methods

As I've hinted at before, security is an aspect-oriented concept. And Spring AOP is the basis for method-level security in Spring Security. But for the most part you'll never need to deal with Spring Security's aspects directly. All of the AOP involved in securing methods is packed into a single element: `<global-method-security>`. Here's a common way of using `<global-method-security>`.

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

This sets up Spring Security for securing methods that are annotated with Spring Security's own `@Secured` annotation. This is just one of four ways that Spring Security supports method-level security:

- Methods annotated with `@Secured`
- Methods annotated with JSR-250's `@RolesAllowed`
- Methods annotated with Spring's pre- and post-invocation annotations
- Methods matching one or more explicitly declared pointcuts

Let's look at each style of method security.

9.5.1 Securing methods with @Secured

When `<global-method-security>` is configured with its `secured-annotations` attribute set to `enabled`, a pointcut is created such that the Spring Security aspects will wrap bean methods that are annotated with `@Secured`. For example:

```
@Secured("ROLE_SPITTER")
public void addSpittle(Spittle spittle) {
  // ...
}
```

The `@Secured` annotation takes an array of `String` as an argument. Each `String` value is a authorization, one of which is required to invoke the method. By passing in `ROLE_SPITTER`, we tell Spring Security to not allow the `saveSpittle()` method to be invoked unless the authenticated user has ROLE_SPITTER as one of their granted authorities.

If more than one value is passed into `@Secured`, then the authenticated user must be granted at least one of those authorities to gain access to the method. For example, the following use of `@Secured` indicates that the user must have ROLE_SPITTER *or* ROLE_ADMIN privilege to invoke the method:

```
@Secured({"ROLE_SPITTER", "ROLE_ADMIN"})
public void addSpittle(Spittle spittle) {
  // ...
}
```

When the method is invoked by an unauthenticated user or by a user not possessing the required privileges, the aspect wrapping the method will throw one of Spring Security's exceptions (probably a subclass of `AuthenticationException` or `AccessDeniedException`). Ultimately the exception will need to be caught. If the secured method is invoked in the course of a web request, the exception will be automatically handled by Spring Security's filters. Otherwise, you'll need to write the code to handle the exception.

The one drawback of the `@Secured` annotation is that it's a Spring-specific annotation. If you're more comfortable using standard annotations, then perhaps you should consider using `@RolesAllowed` instead.

9.5.2 Using JSR-250's @RolesAllowed

The `@RolesAllowed` annotation is equivalent to `@Secured` in almost every way. The only substantial difference is that `@RolesAllowed` is one of Java's standard annotations as defined in JSR-250.[5]

This difference carries more political consequence than technical. But using the standard `@RolesAllowed` annotation may have implications when used in the context of other frameworks or APIs that process that annotation.

Regardless, if you choose to use `@RolesAllowed`, you'll need to turn it on by setting `<global-method-security>`'s `jsr250-annotations` attribute to `enabled`:

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

Although here we've only enabled `jsr250-annotations`, it's good to note that it's not mutually exclusive with `secured-annotations`. These two annotation styles can both be enabled at the same time. And they may even be used side by side with Spring's pre-/post-invocation security annotations, which is what we'll look at next.

9.5.3 Pre-/Post-invocation security with SpEL

Although `@Secured` and `@RolesAllowed` seem to do the trick when it comes to keeping unauthorized users out, that's about all that they can do. Sometimes security constraints are more interesting than just whether a user has privileges or not.

Spring Security 3.0 introduced a handful of new annotations that use SpEL to enable even more interesting security constraints on methods. These new annotations are described in table 9.6.

We'll look at specific examples of each of these in a moment. But first, it's important to know that if you want to use any of these annotations, you'll need to enable them by setting `<global-method-security>`'s `pre-post-annotations` to `enabled`:

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

With the annotations enabled, you can start annotating methods to be secured. Let's start by looking at `@PreAuthorize`.

[5] http://jcp.org/en/jsr/summary?id=250

Table 9.6 Spring Security 3.0 offers four new annotations that can be used to secure methods with SpEL expressions.

Annotations	Description
`@PreAuthorize`	Restricts access to a method before invocation based on the result of evaluating an expression
`@PostAuthorize`	Allows a method to be invoked, but throws a security exception if the expression evaluates to false
`@PostFilter`	Allows a method to be invoked, but filters the results of that method per an expression
`@PreFilter`	Allows a method to be invoked, but filters input prior to entering the method

PRE-AUTHORIZING METHODS

At first glance, `@PreAuthorize` may appear to be nothing more than a SpEL-enabled equivalent to `@Secured` and `@RolesAllowed`. In fact, you could use `@PreAuthorize` to limit access based on the roles given to the authenticated user:

```
@PreAuthorize("hasRole('ROLE_SPITTER')")
public void addSpittle(Spittle spittle) {
  // ...
}
```

The `String` argument to `@PreAuthorize` is a SpEL expression. Here it uses the Spring Security-provided `hasRole()` function to authorize access to the method if the user has the ROLE_SPITTER role.

With SpEL expressions guiding access decisions, far more advanced security constraints can be written. For example, suppose that the average Spitter user can only write spittles of 140 characters or less, but premium users are allowed unlimited spittle lengths. Though `@Secured` and `@RolesAllowed` would be of no help here, `@PreAuthorize` is on the case:

```
@PreAuthorize("(hasRole('ROLE_SPITTER') and #spittle.text.length() <= 140)
               or hasRole('ROLE_PREMIUM')")
public void addSpittle(Spittle spittle) {
  // ...
}
```

The `#spittle` portion of the expression refers directly to the method parameter of the same name. This enables Spring Security to examine the parameters passed to the method and use those parameters in its authorization decision making. Here, we dig into the `Spitter`'s text to make sure it doesn't exceed the length allowed for standard Spitter users. Or if the user is a premium user, then the length doesn't matter.

POST-AUTHORIZING METHODS

A slightly less obvious way to authorize a method is to post-authorize the method. Post-authorization typically involves making security decisions based on the object returned from the secured method. This of course means that the method must be invoked and given a chance to produce a return value.

Aside from the timing of the authorization, `@PostAuthorize` works much the same as `@PreAuthorize`. For example, suppose that we want to secure the `getSpittleById()` method to only authorize access if the `Spittle` object returned belongs to the authenticated user. For that we could annotate `getSpittleById()` with `@PostAuthorize` like this:

```
@PostAuthorize("returnObject.spitter.username == principal.username")
public Spittle getSpittleById(long id) {
  // ...
}
```

For easy access to the object returned from the secured method, Spring Security provides the `returnObject` name in SpEL. Here we know that the returned object is a `Spittle`, so the expression digs into its `spitter` property and pulls the `username` property from that.

On the other side of the double-equal comparison, the expression digs into the built-in `principal` object to get its `username` property. `principal` is another one of Spring Security's special built-in names that represents the principal of the currently authenticated user.

If the `Spittle` object has a `Spitter` whose `username` property is the same as the `principal`'s `username`, the `Spittle` will be returned to the caller. Otherwise, an `AccessDeniedException` will be thrown and the caller won't get to see the `Spittle`.

It's important to keep in mind that, unlike methods annotated with `@PreAuthorize`, `@PostAuthorize`-annotated methods will be executed first and intercepted afterward. That means that care should be taken to make sure that the method doesn't have any side effects that would be undesired if authorization fails.

POST-FILTERING METHODS

Sometimes it's not the method that's being secured, but rather the data being returned from that method. For example, suppose that you wanted to present a list of `Spittle`s to the user, but limit that list to only those `Spittle`s that the user is allowed to delete. In that case, you might annotate the method like this:

```
@PreAuthorize("hasRole('ROLE_SPITTER)")
@PostFilter("filterObject.spitter.username == principal.name")
public List<Spittle> getABunchOfSpittles() {
  ...
}
```

Here, the `@PreAuthorize` annotation only allows users with ROLE_SPITTER authority to execute the method. If the user makes it through that checkpoint, the method will execute and a `List` of `Spittle`s will be returned. But the `@PostFilter` annotation will filter that list, ensuring that the user only sees those `Spittle` objects that belong to the user.

The `filterObject` referenced in the expression refers to an individual element (which we know to be a `Spittle`) in the `List` returned from the method. If that `Spittle`'s `Spitter` has a username that's the same as the authenticated user (the

`principal.name` in the expression), then the element will end up in the filtered list. Otherwise, it'll be left out.

I know what you're thinking. You could write your query such that it only returns `Spittle` objects belonging to our user. That'd be fine if the security rules were such that a user may only delete `Spittle`s that belong to them.

To make things more interesting, let's suppose that in addition to being able to delete a `Spittle` that they own, a user is empowered to delete any `Spittle` that contains profanity. For that, you'll rewrite the `@PostFilter` expression as follows:

```
@PreAuthorize("hasRole('ROLE_SPITTER)")
@PostFilter("hasPermission(filterObject, 'delete')")
public List<Spittle> getSpittlesToDelete() {
  ...
}
```

As used here, the `hasPermission()` operation *should* evaluate to `true` if the user has `delete` permission for the `Spittle` identified by `filterObject`. I say that it *should* evaluate to `true` in that case, but the reality is that by default `hasPermission()` will always return `false`.

If `hasPermission()` always returns `false` by default, then what use is it? Well, the nice thing about a default behavior is that it can be overridden. Overriding the behavior of `hasPermission()` involves creating and registering a permission evaluator. That's what `SpittlePermissionEvaluator` in the following listing is for.

Listing 9.7 A permission evaluator provides the logic behind `hasPermission()`

```
package com.habuma.spitter.security;
import java.io.Serializable;
import org.springframework.security.access.PermissionEvaluator;
import org.springframework.security.core.Authentication;
import com.habuma.spitter.domain.Spittle;

public class SpittlePermissionEvaluator implements PermissionEvaluator {
    public boolean hasPermission(Authentication authentication,
            Object target, Object permission) {
        if (target instanceof Spittle) {
            Spittle spittle = (Spittle) target;
            if ("delete".equals(permission)) {
                return spittle.getSpitter().getUsername().equals(
                    authentication.getName()) || hasProfanity(spittle);
            }
        }
        throw new UnsupportedOperationException(
                "hasPermission not supported for object <" + target
                        + "> and permission <" + permission + ">");
    }

    public boolean hasPermission(Authentication authentication,
            Serializable targetId, String targetType, Object permission) {
        throw new UnsupportedOperationException();
    }
```

```
    private boolean hasProfanity(Spittle spittle) {
        ...
        return false;
    }
}
```

`SpittlePermissionEvaluator` implements Spring Security's `PermissionEvaluator` interface, which demands that two different `hasPermission()` methods be implemented. One of the `hasPermission()` methods takes an `Object` as the object to evaluate against in the second parameter. The other `hasPermission()` is useful when only the ID of the target object is available, and takes that ID as a `Serializable` in its second parameter.

For our purposes, we assume that we'll always have the `Spittle` object to evaluate permissions against, so the other method simply throws `UnsupportedOperationException`.

As for the other `hasPermission()` method, it checks to see that the object being evaluated is a `Spittle` and that we're checking for delete permission. If so, then it compares the `Spitter`'s username against the authenticated user's name. It also checks whether the `Spittle` contains profanity by passing it into the `hasProfanity()` method.[6]

With the permission evaluator ready, you need to register it with Spring Security for it to back the `hasPermission()` operation in the expression given to `@PostFilter`. To do that, you'll need to create an expression handler bean and register it with `<global-method-security>`.

For the expression evaluator, you'll create a bean of type `DefaultMethodSecurityExpressionHandler` and inject its `permissionEvaluator` property with an instance of our `SpittlePermissionEvaluator`:

```
<beans:bean id="expressionHandler" class=
    "org.springframework.security.access.expression.method.
                                    ➥DefaultMethodSecurityExpressionHandler">
  <beans:property name="permissionEvaluator">
    <beans:bean class=
                "com.habuma.spitter.security.SpittlePermissionEvaluator" />
  </beans:property>
</beans:bean>
```

Then we can configure that `expressionHandler` bean with `<global-method-security>` like this:

```
<global-method-security pre-post-annotations="enabled">
    <expression-handler ref="expressionHandler"/>
</global-method-security>
```

Before, we configured a `<global-method-security>` without specifying an expression handler. But here we have replaced the default expression handler with one that knows about our permission evaluator.

---

[6] I've conveniently left the implementation of `hasProfanity()` as an exercise for the reader.

9.5.4 Declaring method-level security pointcuts

Method-level security constraints often vary from method to method. Annotating each method with the constraints that best serve that method makes a lot of sense. But sometimes it may make sense to apply the same authorization checks to several methods—cross-cutting authorization, so to speak.

To restrict access to multiple methods, we can use the `<protect-pointcut>` element as a child of the `<global-method-security>` element. For example:

```
<global-method-security>
  <protect-pointcut access="ROLE_SPITTER"
      expression=
          "execution(@com.habuma.spitter.Sensitive * *.*(String))"/>
</global-method-security>
```

The `expression` attribute is given an AspectJ pointcut expression. In this case, it identifies any methods that are annotated with a custom `@Sensitive` annotation. Meanwhile, the `access` attribute indicates which authorities the authenticated user must have to access the methods that are identified by the `expression` attribute.

9.6 Summary

Security is a crucial aspect of many applications. Spring Security provides a mechanism for securing your application that's simple, flexible, and powerful.

Using a series of servlet filters, Spring Security can control access to web resources, including Spring MVC controllers. And by employing aspects, you can also secure method invocations with Spring Security. But thanks to Spring Security's configuration namespace, you don't need to deal with the filters or aspects directly. Security can be declared concisely.

When it comes to authenticating users, Spring Security offers several options. We saw how to configure authentication against an in-memory user repository, a relational-database, and LDAP directory servers.

Next, we'll look at ways to integrate Spring applications with other applications. Starting in the next chapter, we'll look at how Spring supports several remoting options including RMI and web services.