Spring Security

Reference Documentation

2.0.x

Preface

Spring Security provides a comprehensive security solution for J2EE-based enterprise software applications. As you will discover as you venture through this reference guide, we have tried to provide you a useful and highly configurable security system.

Security is an ever-moving target, and it's important to pursue a comprehensive, system-wide approach. In security circles we encourage you to adopt "layers of security", so that each layer tries to be as secure as possible in its own right, with successive layers providing additional security. The "tighter" the security of each layer, the more robust and safe your application will be. At the bottom level you'll need to deal with issues such as transport security and system identification, in order to mitigate man-in-the-middle attacks. Next you'll generally utilise firewalls, perhaps with VPNs or IP security to ensure only authorised systems can attempt to connect. In corporate environments you may deploy a DMZ to separate public-facing servers from backend database and application servers. Your operating system will also play a critical part, addressing issues such as running processes as non-privileged users and maximising file system security. An operating system will usually also be configured with its own firewall. Hopefully somewhere along the way you'll be trying to prevent denial of service and brute force attacks against the system. An intrusion detection system will also be especially useful for monitoring and responding to attacks, with such systems able to take protective action such as blocking offending TCP/IP addresses in real-time. Moving to the higher layers, your Java Virtual Machine will hopefully be configured to minimize the permissions granted to different Java types, and then your application will add its own problem domain-specific security configuration. Spring Security makes this latter area - application security - much easier.

Of course, you will need to properly address all security layers mentioned above, together with managerial factors that encompass every layer. A non-exhaustive list of such managerial factors would include security bulletin monitoring, patching, personnel vetting, audits, change control, engineering management systems, data backup, disaster recovery, performance benchmarking, load monitoring, centralised logging, incident response procedures etc.

With Spring Security being focused on helping you with the enterprise application security layer, you will find that there are as many different requirements as there are business problem domains. A banking application has different needs from an ecommerce application. An ecommerce application has different needs from a corporate sales force automation tool. These custom requirements make application security interesting, challenging and rewarding.

Please read Part I, “Getting Started”, in its entirety to begin with. This will introduce you to the framework and the namespace-based configuration system with which you can get up and running quite quickly. To get more of an understanding of an in-depth understaning of how Spring Security works, and some of the classes you might need to use, you should then read Part II, “Overall Architecture”. The remaining parts of this guide are structured in a more traditional reference style, designed to be read on an as-required basis. We'd also recommend that you read up as much as possible on application security issues in general. Spring Security is not a panacea which will solve all security issues. It is important that the application is designed with security in mind from the start. Attempting to retrofit it is not a good idea. In particular, if you are building a web application, you should be aware of the many potential vulnerabilities such as cross-site scripting, request-forgery and session-hijacking which you should be taking into account from the start. The OWASP web site (http://www.owasp.org/) maintains a top ten list of web application vulnerabilities as well as a lot of useful reference information.

We hope that you find this reference guide useful, and we welcome your feedback and suggestions.

Finally, welcome to the Spring Security community.

Part I. Getting Started

The later parts of this guide provide an in-depth discussion of the framework architecture and implementation classes, an understanding of which is important if you need to do any serious customization. In this part, we'll introduce Spring Security 2.0, give a brief overview of the project's history and take a slightly gentler look at how to get started using the framework. In particular, we'll look at namespace configuration which provides a much simpler way of securing your application compared to the traditional Spring bean approach where you had to wire up all the implementation classes individually.

We'll also take a look at the sample applications that are available. It's worth trying to run these and experimenting with them a bit even before you read the later sections - you can dip back into them as your understanding of the framework increases.

Chapter 1. Introduction

1.1. What is Spring Security?

Spring Security provides comprehensive security services for J2EE-based enterprise software applications. There is a particular emphasis on supporting projects built using The Spring Framework, which is the leading J2EE solution for enterprise software development. If you're not using Spring for developing enterprise applications, we warmly encourage you to take a closer look at it. Some familiarity with Spring - and in particular dependency injection principles - will help you get up to speed with Spring Security more easily.

People use Spring Security for many reasons, but most are drawn to the project after finding the security features of J2EE's Servlet Specification or EJB Specification lack the depth required for typical enterprise application scenarios. Whilst mentioning these standards, it's important to recognise that they are not portable at a WAR or EAR level. Therefore, if you switch server environments, it is typically a lot of work to reconfigure your application's security in the new target environment. Using Spring Security overcomes these problems, and also brings you dozens of other useful, entirely customisable security features.

As you probably know, security comprises two major operations. The first is known as "authentication", which is the process of establishing a principal is who they claim to be. A "principal" generally means a user, device or some other system which can perform an action in your application. "Authorization" refers to the process of deciding whether a principal is allowed to perform an action in your application. To arrive at the point where an authorization decision is needed, the identity of the principal has already been established by the authentication process. These concepts are common, and not at all specific to Spring Security.

At an authentication level, Spring Security supports a wide range of authentication models. Most of these authentication models are either provided by third parties, or are developed by relevant standards bodies such as the Internet Engineering Task Force. In addition, Spring Security provides its own set of authentication features. Specifically, Spring Security currently supports authentication integration with all of these technologies:

- HTTP BASIC authentication headers (an IEFT RFC-based standard)
- HTTP Digest authentication headers (an IEFT RFC-based standard)
- HTTP X.509 client certificate exchange (an IEFT RFC-based standard)
- LDAP (a very common approach to cross-platform authentication needs, especially in large environments)
- Form-based authentication (for simple user interface needs)
- OpenID authentication
- Computer Associates Siteminder
- JA-SIG Central Authentication Service (otherwise known as CAS, which is a popular open source single sign on system)
- Transparent authentication context propagation for Remote Method Invocation (RMI) and HttpInvoker (a Spring remoting protocol)
- Automatic "remember-me" authentication (so you can tick a box to avoid re-authentication for a predetermined period of time)
- Anonymous authentication (allowing every call to automatically assume a particular security identity)
- Run-as authentication (which is useful if one call should proceed with a different security identity)
- Java Authentication and Authorization Service (JAAS)
- Container integration with JBoss, Jetty, Resin and Tomcat (so you can still use Container Manager Authentication if desired)
- Java Open Source Single Sign On (JOSSO) *
- OpenNMS Network Management Platform *
- AppFuse *

- AndroMDA *
- Mule ESB *
- Direct Web Request (DWR) *
- Grails *
- Tapestry *
- JTrac *
- Jasypt *
- Roller *
- Elastic Plath *
- Atlassian Crowd *
- Your own authentication systems (see below)

(* Denotes provided by a third party; check our integration page for links to the latest details)

Many independent software vendors (ISVs) adopt Spring Security because of this significant choice of flexible authentication models. Doing so allows them to quickly integrate their solutions with whatever their end clients need, without undertaking a lot of engineering or requiring the client to change their environment. If none of the above authentication mechanisms suit your needs, Spring Security is an open platform and it is quite simple to write your own authentication mechanism. Many corporate users of Spring Security need to integrate with "legacy" systems that don't follow any particular security standards, and Spring Security is happy to "play nicely" with such systems.

Sometimes the mere process of authentication isn't enough. Sometimes you need to also differentiate security based on the way a principal is interacting with your application. For example, you might want to ensure requests only arrive over HTTPS, in order to protect passwords from eavesdropping or end users from man-in-the-middle attacks. Or, you might want to ensure that an actual human being is making the requests and not some robot or other automated process. This is especially helpful to protect password recovery processes from brute force attacks, or simply to make it harder for people to duplicate your application's key content. To help you achieve these goals, Spring Security fully supports automatic "channel security", together with JCaptcha integration for human user detection.

Irrespective of how authentication was undertaken, Spring Security provides a deep set of authorization capabilities. There are three main areas of interest in respect of authorization, these being authorizing web requests, authorizing methods can be invoked, and authorizing access to individual domain object instances. To help you understand the differences, consider the authorization capabilities found in the Servlet Specification web pattern security, EJB Container Managed Security and file system security respectively. Spring Security provides deep capabilities in all of these important areas, which we'll explore later in this reference guide.

1.2. History

Spring Security began in late 2003 as "The Acegi Security System for Spring". A question was posed on the Spring Developers' mailing list asking whether there had been any consideration given to a Spring-based security implementation. At the time the Spring community was relatively small (especially by today's size!), and indeed Spring itself had only existed as a SourceForge project from early 2003. The response to the question was that it was a worthwhile area, although a lack of time currently prevented its exploration.

With that in mind, a simple security implementation was built and not released. A few weeks later another member of the Spring community inquired about security, and at the time this code was offered to them. Several other requests followed, and by January 2004 around twenty people were using the code. These pioneering users were joined by others who suggested a SourceForge project was in order, which was duly established in March 2004.

In those early days, the project didn't have any of its own authentication modules. Container Managed Security

was relied upon for the authentication process, with Acegi Security instead focusing on authorization. This was suitable at first, but as more and more users requested additional container support, the fundamental limitation of container-specific authentication realm interfaces was experienced. There was also a related issue of adding new JARs to the container's classpath, which was a common source of end user confusion and misconfiguration.

Acegi Security-specific authentication services were subsequently introduced. Around a year later, Acegi Security became an official Spring Framework subproject. The 1.0.0 final release was published in May 2006 - after more than two and a half years of active use in numerous production software projects and many hundreds of improvements and community contributions.

Acegi Security became an official Spring Portfolio project towards the end of 2007 and was rebranded as "Spring Security".

Today Spring Security enjoys a strong and active open source community. There are thousands of messages about Spring Security on the support forums. There is an active core of developers work who work on the code itself and an active community which also regularly share patches and support their peers.

1.3. Release Numbering

It is useful to understand how Spring Security release numbers work, as it will help you identify the effort (or lack thereof) involved in migrating to future releases of the project. Officially, we use the Apache Portable Runtime Project versioning guidelines, which can be viewed at `http://apr.apache.org/versioning.html`. We quote the introduction contained on that page for your convenience:

“Versions are denoted using a standard triplet of integers: MAJOR.MINOR.PATCH. The basic intent is that MAJOR versions are incompatible, large-scale upgrades of the API. MINOR versions retain source and binary compatibility with older minor versions, and changes in the PATCH level are perfectly compatible, forwards and backwards.”

1.4. Getting the Source

Since Spring Security is an Open Source project, we'd strongly encourage you to check out the source code using subversion. This will give you full access to all the sample applications and you can build the most up to date version of the project easily. Having the source for a project is also a huge help in debugging. Exception stack traces are no longer obscure black-box issues but you can get straight to the line that's causing the problem and work out what's happening. The source is the ultimate documentation for a project and often the simplest place to find out how something actually works.

To obtain the source for the project trunk, use the following subversion command:

```
svn checkout http://acegisecurity.svn.sourceforge.net/svnroot/acegisecurity/spring-security/trunk/
```

You can checkout specific versions from `http://acegisecurity.svn.sourceforge.net/svnroot/acegisecurity/spring-security/tags/`.

Chapter 2. Security Namespace Configuration

2.1. Introduction

Namespace configuration has been available since version 2.0 of the Spring framework. It allows you to supplement the traditional Spring beans application context syntax with elements from additional XML schema. You can find more information in the Spring Reference Documentation. A namespace element can be used simply to allow a more concise way of configuring an individual bean or, more powerfully, to define an alternative configuration syntax which more closely matches the problem domain and hides the underlying complexity from the user. A simple element may conceal the fact that multiple beans and processing steps are being added to the application context. For example, adding the following element from the security namespace to an application context will start up an embedded LDAP server for testing use within the application:

```
<security:ldap-server />
```

This is much simpler than wiring up the equivalent Apache Directory Server beans. The most common alternative configuration requirements are supported by attributes on the `ldap-server` element and the user is isolated from worrying about which beans they need to be set on and what the bean property names are. [1]. Use of a good XML editor while editing the application context file should provide information on the attributes and elements that are available. We would recommend that you try out the SpringSource Tool Suite as it has special features for working with the Spring portfolio namespaces.

To start using the security namespace in your application context, all you need to do is add the schema declaration to your application context 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/sp
              http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spri
    ...
</beans>
```

In many of the examples you will see (and in the sample) applications, we will often use "security" as the default namespace rather than "beans", which means we can omit the prefix on all the security namespace elements, making the context easier to read. You may also want to do this if you have your application context divided up into separate files and have most of your security configuration in one of them. Your security application context file would then start like this

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="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/sp
              http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spri
    ...
</beans:beans>
```

We'll assume this syntax is being used from now on in this chapter.

2.1.1. Design of the Namespace

[1] You can find out more about the use of the `ldap-server` element in the chapter on LDAP.

The namespace is designed to capture the most common uses of the framework and provide a simplified and concise syntax for enabling them within an application. The design is largely based around the large-scale dependencies within the framework, and can be divided up into the following areas:

- *Web/HTTP Security* - the most complex part. Sets up the filters and related service beans used to apply the framework authentication mechanisms, to secure URLs, render login and error pages and much more.
- *Business Object (Method) Security* - options for securing the service layer.
- *AuthenticationManager* - handles authentication requests from other parts of the framework. A default instance will be registered internally by the namespace.
- *AccessDecisionManager* - provides access decisions for web and method security. A default one will be registered, but you can also choose to use a custom one, declared using normal Spring bean syntax.
- *AuthenticationProvider*s - mechanisms against which the authentication manager authenticates users. The namespace provides supports for several standard options and also a means of adding custom beans declared using a traditional syntax.
- *UserDetailsService* - closely related to authentication providers, but often also required by other beans.

We'll see how these work together in the following sections.

2.2. Getting Started with Security Namespace Configuration

In this section, we'll look at how you can build up a namespace configuration to use some of the main features of the framework. Let's assume you initially want to get up and running as quickly as possible and add authentication support and access control to an existing web application, with a few test logins. Then we'll look at how to change over to authenticating against a database or other security information repository. In later sections we'll introduce more advanced namespace configuration options.

2.2.1. `web.xml` Configuration

The first thing you need to do is add the following filter declaration to your `web.xml` file:

```
<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>
```

This provides a hook into the Spring Security web infrastructure. `DelegatingFilterProxy` is a Spring Framework class which delegates to a filter implementation which is defined as a Spring bean in your application context. In this case, the bean is named "springSecurityFilterChain", which is an internal infrastructure bean created by the namespace to handle web security. Note that you should not use this bean name yourself. Once you've added this to your `web.xml`, you're ready to start editing your application context file. Web security services are configured using the `<http>` element.

2.2.2. A Minimal <http> Configuration

All you need to enable web security to begin with is

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

Which says that we want all URLs within our application to be secured, requiring the role `ROLE_USER` to access them.

> **Note**
>
> You can use multiple `<intercept-url>` elements to define different access requirements for different sets of URLs, but they will be evaluated in the order listed and the first match will be used. So you must put the most specific matches at the top.

To add some users, you can define a set of test data directly in the namespace:

```
<authentication-provider>
  <user-service>
    <user name="jimi" password="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob" password="bobspassword" authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
```

If you are familiar with previous versions of the framework, you can probably already guess roughly what's going on here. The <http> element is responsible for creating a `FilterChainProxy` and the filter beans which it uses. Common issues like incorrect filter ordering are no longer an issue as the filter positions are predefined.

The `<authentication-provider>` element creates a `DaoAuthenticationProvider` bean and the `<user-service>` element creates an `InMemoryDaoImpl`. A `ProviderManager` bean is always created by the namespace processing system and the `DaoAuthenticationProvider` is automatically registered with it. You can find more detailed information on the beans that are created in the namespace appendix.

The configuration above defines two users, their passwords and their roles within the application (which will be used for access control). It is also possible to load user information from a standard properties file using the `properties` attribute on `user-service`. See the section on in-memory authentication for more details. Using the `<authentication-provider>` element means that the user information will be used by the authentication manager to process authentication requests.

At this point you should be able to start up your application and you will be required to log in to proceed. Try it out, or try experimenting with the "tutorial" sample application that comes with the project. The above configuration actually adds quite a few services to the application because we have used the `auto-config` attribute. For example, form login processing and "remember-me" services are automatically enabled.

2.2.2.1. What does `auto-config` Include?

The `auto-config` attribute, as we have used it above, is just a shorthand syntax for:

```
<http>
  <intercept-url pattern="/**" access="ROLE_USER" />
  <form-login />
  <anonymous />
  <http-basic />
  <logout />
  <remember-me />
</http>
```

These other elements are responsible for setting up form-login, anonymous authentication, basic authentication, logout handling and remember-me services respectively. They each have attributes which can be used to alter their behaviour.

`auto-config` Requires a UserDetailsService

An error can occur when using auto-config without a `UserDetailsService` in your application context (for example, if you are using LDAP authentication). This is because remember-me is automatically enabled when `auto-config="true"` and it requires an authentication mechanism which uses a `UserDetailsService` to function (see the Remember-me chapter for more details). If you have an error caused by a missing `UserDetailsService` then try removing the `auto-config` setting (and any `remember-me` setting you might have).

2.2.2.2. Form and Basic Login Options

You might be wondering where the login form came from when you were prompted to log in, since we made no mention of any HTML files or JSPs. In fact, since we didn't explicitly set a URL for the login page, Spring Security generates one automatically, based on the features that are enabled and using standard values for the URL which processes the submitted login, the default target URL the user will be sent to and so on. However, the namespace offers plenty of suppport to allow you to customize these options. For example, if you want to supply your own login page, you could use:

```
<http auto-config='true'>
  <intercept-url pattern="/login.jsp*" filters="none"/>
  <intercept-url pattern="/**" access="ROLE_USER" />
  <form-login login-page='/login.jsp'/>
</http>
```

Note that you can still use `auto-config`. The `form-login` element just overrides the default settings. Also note that we've added an extra `intercept-url` element to say that any requests for the login page should be excluded from processing by the security filters. Otherwise the request would be matched by the pattern `/**` and it wouldn't be possible to access the login page itself! If you want to use basic authentication instead of form login, then change the configuration to

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

Basic authentication will then take precedence and will be used to prompt for a login when a user attempts to access a protected resource. Form login is still available in this configuration if you wish to use it, for example

through a login form embedded in another web page.

2.2.2.2.1. Setting a Default Post-Login Destination

If a form login isn't prompted by an attempt to access a protected resource, the `default-target-url` option comes into play. This is the URL the user will be taken to after logging in, and defaults to "/". You can also configure things so that they user *always* ends up at this page (regardless of whether the login was "on-demand" or they explicitly chose to log in) by setting the `always-use-default-target` attribute to "true". This is useful if your application always requires that the user starts at a "home" page, for example:

```
<http>
  <intercept-url pattern='/login.htm*' filters='none'/>
  <intercept-url pattern='/**' access='ROLE_USER' />
  <form-login login-page='/login.htm' default-target-url='/home.htm' always-use-default-target='true' />
</http>
```

2.2.3. Using other Authentication Providers

In practice you will need a more scalable source of user information than a few names added to the application context file. Most likely you will want to store your user information in something like a database or an LDAP server. LDAP namespace configuration is dealt with in the LDAP chapter, so we won't cover it here. If you have a custom implementation of Spring Security's `UserDetailsService`, called "myUserDetailsService" in your application context, then you can authenticate against this using

```
<authentication-provider user-service-ref='myUserDetailsService'/>
```

If you want to use a database, then you can use

```
<authentication-provider>
  <jdbc-user-service data-source-ref="securityDataSource"/>
</authentication-provider>
```

Where "securityDataSource" is the name of a `DataSource` bean in the application context, pointing at a database containing the standard Spring Security user data tables. Alternatively, you could configure a Spring Security `JdbcDaoImpl` bean and point at that using the `user-service-ref` attribute:

```
<authentication-provider user-service-ref='myUserDetailsService'/>

<beans:bean id="myUserDetailsService" class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
  <beans:property name="dataSource" ref="dataSource"/>
</beans:bean>
```

You can also use standard `AuthenticationProvider` beans by adding the `<custom-authentication-provider>` element within the bean definition. See Section 2.6, “The Default Authentication Manager” for more on this.

2.2.3.1. Adding a Password Encoder

Often your password data will be encoded using a hashing algorithm. This is supported by the

<password-encoder> element. With SHA encoded passwords, the original authentication provider configuration would look like this:

```
<authentication-provider>
  <password-encoder hash="sha"/>
  <user-service>
    <user name="jimi" password="d7e6351eaa13189a5a3641bab846c8e8c69ba39f" authorities="ROLE_USER, ROLE_ADMIN" />
    <user name="bob" password="4e7421b1b8765d8f9406d87e7cc6aa784c4ab97f" authorities="ROLE_USER" />
  </user-service>
</authentication-provider>
```

When using hashed passwords, it's also a good idea to use a salt value to protect against dictionary attacks and Spring Security supports this too. Ideally you would want to use a randomly generated salt value for each user, but you can use any property of the UserDetails object which is loaded by your UserDetailsService. For example, to use the username property, you would use

```
<password-encoder hash="sha">
  <salt-source user-property="username"/>
</password-encoder>
```

You can use a custom password encoder bean by using the ref attribute of password-encoder. This should contain the name of a bean in the application context which is an instance of Spring Security's PasswordEncoder interface.

2.3. Advanced Web Features

2.3.1. Remember-Me Authentication

See the separate Remember-Me chapter for information on remember-me namespace configuration.

2.3.2. Adding HTTP/HTTPS Channel Security

If your application supports both HTTP and HTTPS, and you require that particular URLs can only be accessed over HTTPS, then this is directly supported using the requires-channel attribute on <intercept-url>:

```
<http>
  <intercept-url pattern="/secure/**" access="ROLE_USER" requires-channel="https"/>
  <intercept-url pattern="/**" access="ROLE_USER" requires-channel="any"/>
  ...
</http>
```

With this configuration in place, if a user attempts to access anything matching the "/secure/**" pattern using HTTP, they will first be redirected to an HTTPS URL. The available options are "http", "https" or "any". Using the value "any" means that either HTTP or HTTPS can be used.

If your application uses non-standard ports for HTTP and/or HTTPS, you can specify a list of port mappings as follows:

```
<http>
  ...
  <port-mappings>
```

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

You can find a more in-depth discussion of channel security in Chapter 7, *Channel Security*.

2.3.3. Concurrent Session Control

If you wish to place constraints on a single user's ability to log in to your application, Spring Security supports this out of the box with the following simple additions. First you need to add the following listener to your `web.xml` file to keep Spring Security updated about session lifecycle events:

```
<listener>
  <listener-class>org.springframework.security.ui.session.HttpSessionEventPublisher</listener-class>
</listener>
```

Then add the following line to your application context:

```
<http>
  ...
  <concurrent-session-control max-sessions="1" />
</http>
```

This will prevent a user from logging in multiple times - a second login will cause the first to be invalidated. Often you would prefer to prevent a second login, in which case you can use

```
<http>
  ...
  <concurrent-session-control max-sessions="1" exception-if-maximum-exceeded="true"/>
</http>
```

The second login will then be rejected.

2.3.4. OpenID Login

The namespace supports OpenID login either instead of, or in addition to normal form-based login, with a simple change:

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

You should then register yourself with an OpenID provider (such as myopenid.com), and add the user information to your in-memory `<user-service>`:

```
<user name="http://jimi.hendrix.myopenid.com/" password="notused" authorities="ROLE_USER" />
```

You should be able to login using the `myopenid.com` site to authenticate.

2.3.5. Adding in Your Own Filters

If you've used Spring Security before, you'll know that the framework maintains a chain of filters in order to apply its services. You may want to add your own filters to the stack at particular locations or use a Spring Security filter for which there isn't currently a namespace configuration option (CAS, for example). Or you might want to use a customized version of a standard namespace filter, such as the `AuthenticationProcessingFilter` which is created by the `<form-login>` element, taking advantage of some of the extra configuration options which are available by using defining the bean directly. How can you do this with namespace configuration, since the filter chain is not directly exposed?

The order of the filters is always strictly enforced when using the namespace. Each Spring Security filter implements the Spring `Ordered` interface and the filters created by the namespace are sorted during initialization. The standard Spring Security filters each have an alias in the namespace. The filters, aliases and namespace elements/attributes which create the filters are shown in Table 2.1, “Standard Filter Aliases and Ordering”.

Table 2.1. Standard Filter Aliases and Ordering

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	`ChannelProcessingFilter`	`http/intercept-url`
CONCURRENT_SESSION_FILTER	`ConcurrentSessionFilter`	`http/concurrent-session-control`
SESSION_CONTEXT_INTEGRATION_FILTER	`HttpSessionContextIntegrationFilter`	`http`
LOGOUT_FILTER	`LogoutFilter`	`http/logout`
X509_FILTER	`X509PreAuthenticatedProcessingFilter`	`http/x509`
PRE_AUTH_FILTER	`AstractPreAuthenticatedProcessingFilter` Subclasses	N/A
CAS_PROCESSING_FILTER	`CasProcessingFilter`	N/A
AUTHENTICATION_PROCESSING_FILTER	`AuthenticationProcessingFilter`	`http/form-login`
BASIC_PROCESSING_FILTER	`BasicProcessingFilter`	`http/http-basic`
SERVLET_API_SUPPORT_FILTER	`SecurityContextHolderAwareRequestFilter`	`http/servlet-api-provision`
REMEMBER_ME_FILTER	`RememberMeProcessingFilter`	`http/remember-me`
ANONYMOUS_FILTER	`AnonymousProcessingFilter`	`http/anonymous`
EXCEPTION_TRANSLATION_FILTER	`ExceptionTranslationFilter`	`http`
NTLM_FILTER	`NtlmProcessingFilter`	N/A
FILTER_SECURITY_INTERCEPTOR	`FilterSecurityInterceptor`	`http`
SWITCH_USER_FILTER	`SwitchUserProcessingFilter`	N/A

You can add your own filter to the stack, using the `custom-filter` element and one of these names to specify the position your filter should appear at:

```
<beans:bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter">
  <custom-filter position="AUTHENTICATION_PROCESSING_FILTER"/>
</beans:bean>
```

You can also use the `after` or `before` attribtues if you want your filter to be inserted before or after another filter in the stack. The names "FIRST" and "LAST" can be used with the `position` attribute to indicate that you want your filter to appear before or after the entire stack, respectively.

Avoiding filter position conflicts

If you are inserting a custom filter which may occupy the same position as one of the standard filters created by the namespace then it's important that you don't include the namespace versions by mistake. Avoid using the `auto-config` attribute and remove any elements which create filters whose functionality you want to replace.

Note that you can't replace filters which are created by the use of the `<http>` element itself - `HttpSessionContextIntegrationFilter`, `ExceptionTranslationFilter` or `FilterSecurityInterceptor`.

If you're replacing a namespace filter which requires an authentication entry point (i.e. where the authentication process is triggered by an attempt by an unauthenticated user to access to a secured resource), you will need to add a custom entry point bean too.

2.3.5.1. Setting a Custom `AuthenticationEntryPoint`

If you aren't using form login, OpenID or basic authentication through the namespace, you may want to define an authentication filter and entry point using a traditional bean syntax and link them into the namespace, as we've just seen. The corresponding `AuthenticationEntryPoint` can be set using the `entry-point-ref` attribute on the `<http>` element.

The CAS sample application is a good example of the use of custom beans with the namespace, including this syntax. If you aren't familiar with authentication entry points, they are discussed in the technical overview chapter.

2.3.6. Session Fixation Attack Protection

Session fixation attacks are a potential risk where it is possible for a malicious attacker to create a session by accessing a site, then persuade another user to log in with the same session (by sending them a link containing the session identifier as a parameter, for example). Spring Security protects against this automatically by creating a new session when a user logs in. If you don't require this protection, or it conflicts with some other requirement, you can control the behaviour using the `session-fixation-protection` attribute on `<http>`, which has three options

- `migrateSession` - creates a new session and copies the existing session attributes to the new session. This is the default.
- `none` - Don't do anything. The original session will be retained.
- `newSession` - Create a new "clean" session, without copying the existing session data.

2.4. Method Security

Spring Security 2.0 has improved support substantially for adding security to your service layer methods. If you are using Java 5 or greater, then support for JSR-250 security annotations is provided, as well as the framework's native `@Secured` annotation. You can apply security to a single bean, using the

intercept-methods element to decorate the bean declaration, or you can secure multiple beans across the entire service layer using the AspectJ style pointcuts.

2.4.1. The <global-method-security> Element

This element is used to enable annotation-based security in your application (by setting the appropriate attributes on the element), and also to group together security pointcut declarations which will be applied across your entire application context. You should only declare one <global-method-security> element. The following declaration would enable support for both Spring Security's @Secured, and JSR-250 annotations:

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

Adding an annotation to a method (on an class or interface) would then limit the access to that method accordingly. Spring Security's native annotation support defines a set of attributes for the method. These will be passed to the AccessDecisionManager for it to make the actual decision. This example is taken from the tutorial sample, which is a good starting point if you want to use method security in your application:

```
public interface BankService {

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account readAccount(Long id);

  @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
  public Account[] findAccounts();

  @Secured("ROLE_TELLER")
  public Account post(Account account, double amount);
}
```

2.4.1.1. Adding Security Pointcuts using protect-pointcut

The use of protect-pointcut is particularly powerful, as it allows you to apply security to many beans with only a simple declaration. Consider the following example:

```
<global-method-security>
  <protect-pointcut expression="execution(* com.mycompany.*Service.*(..))" access="ROLE_USER"/>
</global-method-security>
```

This will protect all methods on beans declared in the application context whose classes are in the com.mycompany package and whose class names end in "Service". Only users with the ROLE_USER role will be able to invoke these methods. As with URL matching, the most specific matches must come first in the list of pointcuts, as the first matching expression will be used.

2.4.2. The intercept-methods Bean Decorator

This alternative syntax allows you to specify security for a specific bean by adding this element within the bean itself.

```
<bean:bean id="target" class="com.mycompany.myapp.MyBean">
    <intercept-methods>
        <protect method="set*" access="ROLE_ADMIN" />
        <protect method="get*" access="ROLE_ADMIN,ROLE_USER" />
        <protect method="doSomething" access="ROLE_USER" />
    </intercept-methods>
</bean:bean>
```

```
```

This allows you to configure security attributes for individual methods on the bean or simple wildcarded patterns.

2.5. The Default AccessDecisionManager

This section assumes you have some knowledge of the underlying architecture for access-control within Spring Security. If you don't you can skip it and come back to it later, as this section is only really relevant for people who need to do some customization in order to use more than simple role based security.

When you use a namespace configuration, a default instance of `AccessDecisionManager` is automatically registered for you and will be used for making access decisions for method invocations and web URL access, based on the access attributes you specify in your `intercept-url` and `protect-pointcut` declarations (and in annotations if you are using annotation secured methods).

The default strategy is to use an `AffirmativeBased AccessDecisionManager` with a `RoleVoter` and an `AuthenticatedVoter`.

2.5.1. Customizing the AccessDecisionManager

If you need to use a more complicated access control strategy then it is easy to set an alternative for both method and web security.

For method security, you do this by setting the `access-decision-manager-ref` attribute on `global-method-security`to the Id of the appropriate `AccessDecisionManager` bean in the application context:

```
<global-method-security access-decision-manager-ref="myAccessDecisionManagerBean">
  ...
</global-method-security>
```

The syntax for web security is the same, but on the `http` element:

```
<http access-decision-manager-ref="myAccessDecisionManagerBean">
  ...
</http>
```

2.6. The Default Authentication Manager

We've touched on the idea that the namespace configuration automatically registers an authentication manager bean for you. This is an instance of Spring Security's `ProviderManager` class, which you may already be familiar with if you've used the framework before. You can't use a custom `AuthenticationProvider` if you are using either HTTP or method security through the namespace, but this should not be a problem as you have full control over the `AuthenticationProviders` that are used.

You may want to register additional `AuthenticationProvider` beans with the `ProviderManager` and you can do this using the `<custom-authentication-provider>` element within the bean. For example:

```
<bean id="casAuthenticationProvider"
    class="org.springframework.security.providers.cas.CasAuthenticationProvider">
```

```
  <security:custom-authentication-provider />
  ...
</bean>
```

Another common requirement is that another bean in the context may require a reference to the `AuthenticationManager`. There is a special element which lets you register an alias for the `AuthenticationManager` and you can then use this name elsewhere in your application context.

```
<security:authentication-manager alias="authenticationManager"/>

<bean id="customizedFormLoginFilter" class="org.springframework.security.ui.webapp.AuthenticationProcessingFil
   <security:custom-filter position="AUTHENTICATION_PROCESSING_FILTER "/>
   <property name="authenticationManager" ref="authenticationManager"/>
   ...
</bean>
```

Chapter 3. Sample Applications

There are several sample web applications that are available with the project. To avoid an overly large download, only the "tutorial" and "contacts" samples are included in the distribution zip file. You can either build the others yourself, or you can obtain the war files individually from the central Maven repository. We'd recommend the former. You can get the source as described in the introduction and it's easy to build the project using Maven. There is more information on the project web site at http://www.springframework.org/spring-security/ if you need it. All paths referred to in this chapter are relative to the source directory, once you have checked it out from subversion.

3.1. Tutorial Sample

The tutorial sample is a nice basic example to get you started. It uses simple namespace configuration throughout. The compiled application is included in the distribution zip file, ready to be deployed into your web container (`spring-security-samples-tutorial-2.0.x.war`). The form-based authentication mechanism is used in combination with the commonly-used remember-me authentication provider to automatically remember the login using cookies.

We recommend you start with the tutorial sample, as the XML is minimal and easy to follow. Most importantly, you can easily add this one XML file (and its corresponding `web.xml` entries) to your existing application. Only when this basic integration is achieved do we suggest you attempt adding in method authorization or domain object security.

3.2. Contacts

The Contacts Sample is quite an advanced example in that it illustrates the more powerful features of domain object access control lists in addition to basic application security.

To deploy, simply copy the WAR file from Spring Security distribution into your container's `webapps` directory. The war should be called `spring-security-samples-contacts-2.0.0.war` (the appended version number will vary depending on what release you are using).

After starting your container, check the application can load. Visit `http://localhost:8080/contacts` (or whichever URL is appropriate for your web container and the WAR you deployed).

Next, click "Debug". You will be prompted to authenticate, and a series of usernames and passwords are suggested on that page. Simply authenticate with any of these and view the resulting page. It should contain a success message similar to the following:

```
Authentication object is of type: org.springframework.security.providers.UsernamePasswordAuthenticationToken

Authentication object as a String:
  org.springframework.security.providers.UsernamePasswordAuthenticationToken@1f127853:
  Principal: org.springframework.security.userdetails.User@b07ed00:
  Username: rod; Password: [PROTECTED]; Enabled: true; AccountNonExpired: true;
  credentialsNonExpired: true; AccountNonLocked: true;
  Granted Authorities: ROLE_SUPERVISOR, ROLE_USER; Password: [PROTECTED]; Authenticated: true;
  Details: org.springframework.security.ui.WebAuthenticationDetails@0:
  RemoteIpAddress: 127.0.0.1; SessionId: k5qypsawgpwb;
```

```
 Granted Authorities: ROLE_SUPERVISOR, ROLE_USER

Authentication object holds the following granted authorities:

ROLE_SUPERVISOR (getAuthority(): ROLE_SUPERVISOR)
ROLE_USER (getAuthority(): ROLE_USER)

SUCCESS! Your web filters appear to be properly configured!
```

Once you successfully receive the above message, return to the sample application's home page and click "Manage". You can then try out the application. Notice that only the contacts available to the currently logged on user are displayed, and only users with `ROLE_SUPERVISOR` are granted access to delete their contacts. Behind the scenes, the `MethodSecurityInterceptor` is securing the business objects.

The application allows you to modify the access control lists associated with different contacts. Be sure to give this a try and understand how it works by reviewing the application context XML files.

3.3. LDAP Sample

The LDAP sample application provides a basic configuration and sets up both a namespace configuration and an equivalent configuration using traditional beans, both in the same application context file. This means there are actually two identical authentication providers configured in this application.

3.4. CAS Sample

The CAS sample requires that you run both a CAS server and CAS client. It isn't included in the distribution so you should check out the project code as described in the introduction. You'll find the relevant files under the `sample/cas` directory. There's also a `Readme.txt` file in there which explains how to run both the server and the client directly from the source tree, complete with SSL support. You have to download the CAS Server web application (a war file) from the CAS site and drop it into the `samples/cas/server` directory.

3.5. Pre-Authentication Sample

This sample application demonstrates how to wire up beans from the pre-authentication framework to make use of login information from a J2EE container. The user name and roles are those setup by the container.

The code is in `samples/preauth` .

Chapter 4. Spring Security Community

4.1. Issue Tracking

Spring Security uses JIRA to manage bug reports and enhancement requests. If you find a bug, please log a report using JIRA. Do not log it on the support forum, mailing list or by emailing the project's developers. Such approaches are ad-hoc and we prefer to manage bugs using a more formal process.

If possible, in your issue report please provide a JUnit test that demonstrates any incorrect behaviour. Or, better yet, provide a patch that corrects the issue. Similarly, enhancements are welcome to be logged in the issue tracker, although we only accept enhancement requests if you include corresponding unit tests. This is necessary to ensure project test coverage is adequately maintained.

You can access the issue tracker at http://jira.springframework.org/browse/SEC.

4.2. Becoming Involved

We welcome your involvement in Spring Security project. There are many ways of contributing, including reading the forum and responding to questions from other people, writing new code, improving existing code, assisting with documentation, developing samples or tutorials, or simply making suggestions.

4.3. Further Information

Questions and comments on Spring Security are welcome. You can use the Spring Community Forum web site at `http://forum.springframework.org` to discuss Spring Security with other users of the framework. Remember to use JIRA for bug reports, as explained above. Everyone is also welcome to join the Acegisecurity-developer mailing list and participate in design discussions. The traffic volume is very light.

Part II. Overall Architecture

Like most software, Spring Security has certain central interfaces, classes and conceptual abstractions that are commonly used throughout the framework. In this part of the reference guide we will introduce Spring Security, before examining these central elements that are necessary to successfully planning and executing a Spring Security integration.

Chapter 5. Technical Overview

5.1. Runtime Environment

Spring Security is written to execute within a standard Java 1.4 Runtime Environment. It also supports Java 5.0, although the Java types which are specific to this release are packaged in a separate package with the suffix "tiger" in their JAR filename. As Spring Security aims to operate in a self-contained manner, there is no need to place any special configuration files into your Java Runtime Environment. In particular, there is no need to configure a special Java Authentication and Authorization Service (JAAS) policy file or place Spring Security into common classpath locations.

Similarly, if you are using an EJB Container or Servlet Container there is no need to put any special configuration files anywhere, nor include Spring Security in a server classloader.

This design offers maximum deployment time flexibility, as you can simply copy your target artifact (be it a JAR, WAR or EAR) from one system to another and it will immediately work.

5.2. Shared Components

Let's explore some of the most important shared components in Spring Security. Components are considered "shared" if they are central to the framework and the framework cannot operate without them. These Java types represent the building blocks of the remaining system, so it's important to understand that they're there, even if you don't need to directly interact with them.

5.2.1. SecurityContextHolder, SecurityContext and Authentication Objects

The most fundamental object is `SecurityContextHolder`. This is where we store details of the present security context of the application, which includes details of the principal currently using the application. By default the `SecurityContextHolder` uses a `ThreadLocal` to store these details, which means that the security context is always available to methods in the same thread of execution, even if the security context is not explicitly passed around as an argument to those methods. Using a `ThreadLocal` in this way is quite safe if care is taken to clear the thread after the present principal's request is processed. Of course, Spring Security takes care of this for you automatically so there is no need to worry about it.

Some applications aren't entirely suitable for using a `ThreadLocal`, because of the specific way they work with threads. For example, a Swing client might want all threads in a Java Virtual Machine to use the same security context. For this situation you would use the `SecurityContextHolder.MODE_GLOBAL`. Other applications might want to have threads spawned by the secure thread also assume the same security identity. This is achieved by using `SecurityContextHolder.MODE_INHERITABLETHREADLOCAL`. You can change the mode from the default `SecurityContextHolder.MODE_THREADLOCAL` in two ways. The first is to set a system property. Alternatively, call a static method on `SecurityContextHolder`. Most applications won't need to change from the default, but if you do, take a look at the JavaDocs for `SecurityContextHolder` to learn more.

Inside the `SecurityContextHolder` we store details of the principal currently interacting with the application. Spring Security uses an `Authentication` object to represent this information. Whilst you won't normally need to create an `Authentication` object yourself, it is fairly common for users to query the `Authentication` object. You can use the following code block - from anywhere in your application - to obtain the name of the authenticated user, for example:

```
Object obj = SecurityContextHolder.getContext().getAuthentication().getPrincipal();

if (obj instanceof UserDetails) {
  String username = ((UserDetails)obj).getUsername();
} else {
  String username = obj.toString();
}
```

The above code introduces a number of interesting relationships and key objects. First, you will notice that there is an intermediate object between `SecurityContextHolder` and `Authentication`. The `SecurityContextHolder.getContext()` method is actually returning a `SecurityContext`.

5.2.2. The UserDetailsService

Another item to note from the above code fragment is that you can obtain a principal from the `Authentication` object. The principal is just an `Object`. Most of the time this can be cast into a `UserDetails` object. `UserDetails` is a central interface in Spring Security. It represents a principal, but in an extensible and application-specific way. Think of `UserDetails` as the adapter between your own user database and what Spring Security needs inside the `SecurityContextHolder`. Being a representation of something from your own user database, quite often you will cast the `UserDetails` to the original object that your application provided, so you can call business-specific methods (like `getEmail()`, `getEmployeeNumber()` and so on).

By now you're probably wondering, so when do I provide a `UserDetails` object? How do I do that? I thought you said this thing was declarative and I didn't need to write any Java code - what gives? The short answer is that there is a special interface called `UserDetailsService`. The only method on this interface accepts a `String`-based username argument and returns a `UserDetails`. Most authentication providers that ship with Spring Security delegate to a `UserDetailsService` as part of the authentication process. The `UserDetailsService` is used to build the `Authentication` object that is stored in the `SecurityContextHolder`. The good news is that we provide a number of `UserDetailsService` implementations, including one that uses an in-memory map and another that uses JDBC. Most users tend to write their own, though, with such implementations often simply sitting on top of an existing Data Access Object (DAO) that represents their employees, customers, or other users of the enterprise application. Remember the advantage that whatever your UserDetailsService returns can always be obtained from the `SecurityContextHolder`, as per the above code fragment.

5.2.3. GrantedAuthority

Besides the principal, another important method provided by `Authentication` is `getAuthorities()`. This method provides an array of `GrantedAuthority` objects. A `GrantedAuthority` is, not surprisingly, an authority that is granted to the principal. Such authorities are usually "roles", such as `ROLE_ADMINISTRATOR` or `ROLE_HR_SUPERVISOR`. These roles are later on configured for web authorization, method authorization and domain object authorization. Other parts of Spring Security are capable of interpreting these authorities, and expect them to be present. `GrantedAuthority` objects are usually loaded by the `UserDetailsService`.

Usually the `GrantedAuthority` objects are application-wide permissions. They are not specific to a given domain object. Thus, you wouldn't likely have a `GrantedAuthority` to represent a permission to `Employee` object number 54, because if there are thousands of such authorities you would quickly run out of memory (or, at the very least, cause the application to take a long time to authenticate a user). Of course, Spring Security is expressly designed to handle this common requirement, but you'd instead use the project's domain object security capabilities for this purpose.

Last but not least, sometimes you will need to store the `SecurityContext` between HTTP requests. Other times the principal will re-authenticate on every request, although most of the time it will be stored. The

`HttpSessionContextIntegrationFilter` is responsible for storing a `SecurityContext` between HTTP requests. As suggested by the name of the class, the `HttpSession` is used to store this information. You should never interact directly with the `HttpSession` for security purposes. There is simply no justification for doing so - always use the `SecurityContextHolder` instead.

5.2.4. Summary

Just to recap, the major building blocks of Spring Security are:

- `SecurityContextHolder`, to provide any type access to the `SecurityContext`.
- `SecurityContext`, to hold the `Authentication` and possibly request-specific security information.
- `HttpSessionContextIntegrationFilter`, to store the `SecurityContext` in the `HttpSession` between web requests.
- `Authentication`, to represent the principal in a Spring Security-specific manner.
- `GrantedAuthority`, to reflect the application-wide permissions granted to a principal.
- `UserDetails`, to provide the necessary information to build an Authentication object from your application's DAOs.
- `UserDetailsService`, to create a `UserDetails` when passed in a `String`-based username (or certificate ID or alike).

Now that you've gained an understanding of these repeatedly-used components, let's take a closer look at the process of authentication.

5.3. Authentication

As mentioned in the beginning of this reference guide, Spring Security can participate in many different authentication environments. Whilst we recommend people use Spring Security for authentication and not integrate with existing Container Managed Authentication, it is nevertheless supported - as is integrating with your own proprietary authentication system. Let's first explore authentication from the perspective of Spring Security managing web security entirely on its own, which is illustrative of the most complex and most common situation.

Consider a typical web application's authentication process:

1. You visit the home page, and click on a link.
2. A request goes to the server, and the server decides that you've asked for a protected resource.
3. As you're not presently authenticated, the server sends back a response indicating that you must authenticate. The response will either be an HTTP response code, or a redirect to a particular web page.
4. Depending on the authentication mechanism, your browser will either redirect to the specific web page so that you can fill out the form, or the browser will somehow retrieve your identity (eg a BASIC authentication dialogue box, a cookie, a X509 certificate etc).
5. The browser will send back a response to the server. This will either be an HTTP POST containing the contents of the form that you filled out, or an HTTP header containing your authentication details.
6. Next the server will decide whether or not the presented credentials are valid. If they're valid, the next step will happen. If they're invalid, usually your browser will be asked to try again (so you return to step two above).
7. The original request that you made to cause the authentication process will be retried. Hopefully you've

authenticated with sufficient granted authorities to access the protected resource. If you have sufficient access, the request will be successful. Otherwise, you'll receive back an HTTP error code 403, which means "forbidden".

Spring Security has distinct classes responsible for most of the steps described above. The main participants (in the order that they are used) are the `ExceptionTranslationFilter`, an `AuthenticationEntryPoint`, an authentication mechanism, and an `AuthenticationProvider`.

5.3.1. ExceptionTranslationFilter

`ExceptionTranslationFilter` is a Spring Security filter that has responsibility for detecting any Spring Security exceptions that are thrown. Such exceptions will generally be thrown by an `AbstractSecurityInterceptor`, which is the main provider of authorization services. We will discuss `AbstractSecurityInterceptor` in the next section, but for now we just need to know that it produces Java exceptions and knows nothing about HTTP or how to go about authenticating a principal. Instead the `ExceptionTranslationFilter` offers this service, with specific responsibility for either returning error code 403 (if the principal has been authenticated and therefore simply lacks sufficient access - as per step seven above), or launching an `AuthenticationEntryPoint` (if the principal has not been authenticated and therefore we need to go commence step three).

5.3.2. AuthenticationEntryPoint

The `AuthenticationEntryPoint` is responsible for step three in the above list. As you can imagine, each web application will have a default authentication strategy (well, this can be configured like nearly everything else in Spring Security, but let's keep it simple for now). Each major authentication system will have its own `AuthenticationEntryPoint` implementation, which takes actions such as described in step three.

After your browser decides to submit your authentication credentials (either as an HTTP form post or HTTP header) there needs to be something on the server that "collects" these authentication details. By now we're at step six in the above list. In Spring Security we have a special name for the function of collecting authentication details from a user agent (usually a web browser), and that name is "authentication mechanism". After the authentication details are collected from the user agent, an "`Authentication` request" object is built and then presented to an `AuthenticationProvider`.

5.3.3. AuthenticationProvider

The last player in the Spring Security authentication process is an `AuthenticationProvider`. Quite simply, it is responsible for taking an `Authentication` request object and deciding whether or not it is valid. The provider will either throw an exception or return a fully populated `Authentication` object. Remember our good friends, `UserDetails` and `UserDetailsService`? If not, head back to the previous section and refresh your memory. Most `AuthenticationProviders` will ask a `UserDetailsService` to provide a `UserDetails` object. As mentioned earlier, most application will provide their own `UserDetailsService`, although some will be able to use the JDBC or in-memory implementation that ships with Spring Security. The resultant `UserDetails` object - and particularly the `GrantedAuthority[]`s contained within the `UserDetails` object - will be used when building the fully populated `Authentication` object.

After the authentication mechanism receives back the fully-populated `Authentication` object, it will deem the request valid, put the `Authentication` into the `SecurityContextHolder`, and cause the original request to be retried (step seven above). If, on the other hand, the `AuthenticationProvider` rejected the request, the authentication mechanism will ask the user agent to retry (step two above).

5.3.4. Setting the SecurityContextHolder Contents Directly

Whilst this describes the typical authentication workflow, the good news is that Spring Security doesn't mind how you put an `Authentication` inside the `SecurityContextHolder`. The only critical requirement is that the `SecurityContextHolder` contains an `Authentication` that represents a principal before the `AbstractSecurityInterceptor` needs to authorize a request.

You can (and many users do) write their own filters or MVC controllers to provide interoperability with authentication systems that are not based on Spring Security. For example, you might be using Container-Managed Authentication which makes the current user available from a ThreadLocal or JNDI location. Or you might work for a company that has a legacy proprietary authentication system, which is a corporate "standard" over which you have little control. In such situations it's quite easy to get Spring Security to work, and still provide authorization capabilities. All you need to do is write a filter (or equivalent) that reads the third-party user information from a location, build a Spring Security-specific Authentication object, and put it onto the SecurityContextHolder. It's quite easy to do this, and it is a fully-supported integration approach.

5.4. Secure Objects

Spring Security uses the term "secure object" to refer to any object that can have security (such as an authorization decision) applied to it. The most common examples are method invocations and web requests.

5.4.1. Security and AOP Advice

If you're familiar with AOP, you'd be aware there are different types of advice available: before, after, throws and around. An around advice is very useful, because an advisor can elect whether or not to proceed with a method invocation, whether or not to modify the response, and whether or not to throw an exception. Spring Security provides an around advice for method invocations as well as web requests. We achieve an around advice for method invocations using Spring's standard AOP support and we achieve an around advice for web requests using a standard Filter.

For those not familiar with AOP, the key point to understand is that Spring Security can help you protect method invocations as well as web requests. Most people are interested in securing method invocations on their services layer. This is because the services layer is where most business logic resides in current-generation J2EE applications (for clarification, the author disapproves of this design and instead advocates properly encapsulated domain objects together with the DTO, assembly, facade and transparent persistence patterns, but as use of anemic domain objects is the present mainstream approach, we'll talk about it here). If you just need to secure method invocations to the services layer, Spring's standard AOP (otherwise known as AOP Alliance) will be adequate. If you need to secure domain objects directly, you will likely find that AspectJ is worth considering.

You can elect to perform method authorization using AspectJ or Spring AOP, or you can elect to perform web request authorization using filters. You can use zero, one, two or three of these approaches together. The mainstream usage is to perform some web request authorization, coupled with some Spring AOP method invocation authorization on the services layer.

5.4.2. AbstractSecurityInterceptor

Each secure object type supported by Spring Security has its own class, which is a subclass of `AbstractSecurityInterceptor`. Importantly, by the time the `AbstractSecurityInterceptor` is called, the `SecurityContextHolder` will contain a valid `Authentication` if the principal has been authenticated.

`AbstractSecurityInterceptor` provides a consistent workflow for handling secure object requests, typically:

1. Look up the "configuration attributes" associated with the present request
2. Submitting the secure object, current `Authentication` and configuration attributes to the `AccessDecisionManager` for an authorization decision
3. Optionally change the `Authentication` under which the invocation takes place
4. Allow the secure object to proceed (assuming access was granted)
5. Call the `AfterInvocationManager` if configured, once the invocation has returned.

5.4.2.1. What are Configuration Attributes?

A "configuration attribute" can be thought of as a String that has special meaning to the classes used by `AbstractSecurityInterceptor`. They may be simple role names or have more complex meaning, depending on the how sophisticated the `AccessDecisionManager` implementation is. The `AbstractSecurityInterceptor` is configured with an `ObjectDefinitionSource` which it uses to look up the attributes for a secure object. Usually this configuration will be hidden from the user. Configuration attributes will be entered as annotations on secured methods, or as access attributes on secured URLs (using the namespace `<intercept-url>` syntax).

5.4.2.2. RunAsManager

Assuming `AccessDecisionManager` decides to allow the request, the `AbstractSecurityInterceptor` will normally just proceed with the request. Having said that, on rare occasions users may want to replace the `Authentication` inside the `SecurityContext` with a different `Authentication`, which is handled by the `AccessDecisionManager` calling a `RunAsManager`. This might be useful in reasonably unusual situations, such as if a services layer method needs to call a remote system and present a different identity. Because Spring Security automatically propagates security identity from one server to another (assuming you're using a properly-configured RMI or HttpInvoker remoting protocol client), this may be useful.

5.4.2.3. AfterInvocationManager

Following the secure object proceeding and then returning - which may mean a method invocation completing or a filter chain proceeding - the `AbstractSecurityInterceptor` gets one final chance to handle the invocation. At this stage the `AbstractSecurityInterceptor` is interested in possibly modifying the return object. We might want this to happen because an authorization decision couldn't be made "on the way in" to a secure object invocation. Being highly pluggable, `AbstractSecurityInterceptor` will pass control to an `AfterInvocationManager` to actually modify the object if needed. This class can even entirely replace the object, or throw an exception, or not change it in any way.

`AbstractSecurityInterceptor` and its related objects are shown in Figure 5.1, “The key "secure object" model”.

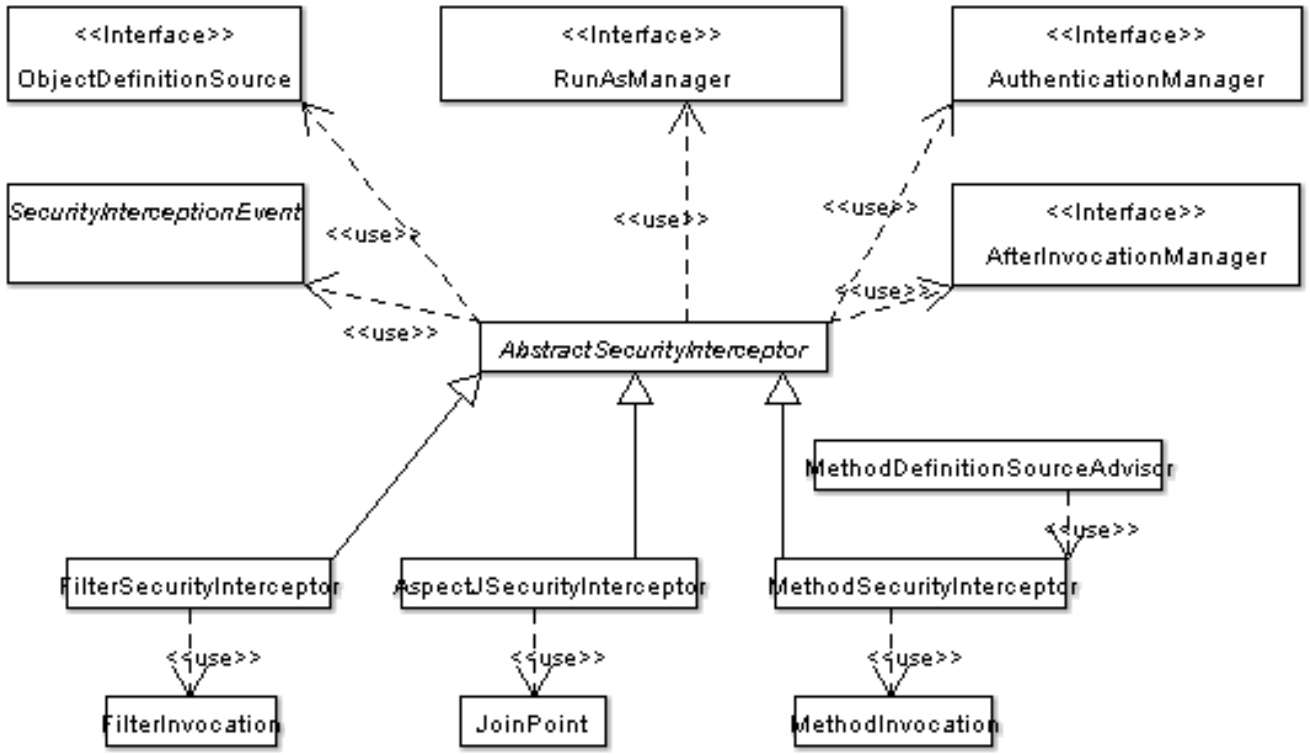

Figure 5.1. The key "secure object" model

5.4.2.4. Extending the Secure Object Model

Only developers contemplating an entirely new way of intercepting and authorizing requests would need to use secure objects directly. For example, it would be possible to build a new secure object to secure calls to a messaging system. Anything that requires security and also provides a way of intercepting a call (like the AOP around advice semantics) is capable of being made into a secure object. Having said that, most Spring applications will simply use the three currently supported secure object types (AOP Alliance `MethodInvocation`, AspectJ `JoinPoint` and web request `FilterInvocation`) with complete transparency.

Chapter 6. Supporting Infrastructure

This chapter introduces some of the supplementary and supporting infrastructure used by Spring Security. If a capability is not directly related to security, yet included in the Spring Security project, we will discuss it in this chapter.

6.1. Localization

Spring Security supports localization of exception messages that end users are likely to see. If your application is designed for English users, you don't need to do anything as by default all Security Security messages are in English. If you need to support other locales, everything you need to know is contained in this section.

All exception messages can be localized, including messages related to authentication failures and access being denied (authorization failures). Exceptions and logging that is focused on developers or system deployers (including incorrect attributes, interface contract violations, using incorrect constructors, startup time validation, debug-level logging) etc are not localized and instead are hard-coded in English within Spring Security's code.

Shipping in the `spring-security-core-xx.jar` you will find an `org.springframework.security` package that in turn contains a `messages.properties` file. This should be referred to by your `ApplicationContext`, as Spring Security classes implement Spring's `MessageSourceAware` interface and expect the message resolver to be dependency injected at application context startup time. Usually all you need to do is register a bean inside your application context to refer to the messages. An example is shown below:

```
<bean id="messageSource" class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
  <property name="basename" value="org/springframework/security/messages"/>
</bean>
```

The `messages.properties` is named in accordance with standard resource bundles and represents the default language supported by Spring Security messages. This default file is in English. If you do not register a message source, Spring Security will still work correctly and fallback to hard-coded English versions of the messages.

If you wish to customize the `messages.properties` file, or support other languages, you should copy the file, rename it accordingly, and register it inside the above bean definition. There are not a large number of message keys inside this file, so localization should not be considered a major initiative. If you do perform localization of this file, please consider sharing your work with the community by logging a JIRA task and attaching your appropriately-named localized version of `messages.properties`.

Rounding out the discussion on localization is the Spring `ThreadLocal` known as `org.springframework.context.i18n.LocaleContextHolder`. You should set the `LocaleContextHolder` to represent the preferred `Locale` of each user. Spring Security will attempt to locate a message from the message source using the `Locale` obtained from this `ThreadLocal`. Please refer to Spring documentation for further details on using `LocaleContextHolder` and the helper classes that can automatically set it for you (eg `AcceptHeaderLocaleResolver`, `CookieLocaleResolver`, `FixedLocaleResolver`, `SessionLocaleResolver` etc)

6.2. Filters

Spring Security uses many filters, as referred to throughout the remainder of this reference guide. If you are

using namespace configuration, then the you don't usually have to declare the filter beans explicitly. There may be times when you want full control over the security filter chain, either because you are using features which aren't supported in the namespace, or you are using your own customized versions of classes.

In this case, you have a choice in how these filters are added to your web application, in that you can use either Spring's `DelegatingFilterProxy` or `FilterChainProxy`. We'll look at both below.

When using `DelegatingFilterProxy`, you will see something like this in the web.xml file:

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

    <filter-mapping>
      <filter-name>myFilter</filter-name>
      <url-pattern>/*</url-pattern>
    </filter-mapping>
```

Notice that the filter is actually a `DelegatingFilterProxy`, and not the filter that will actually implement the logic of the filter. What `DelegatingFilterProxy` does is delegate the `Filter`'s methods through to a bean which is obtained from the Spring application context. This enables the bean to benefit from the Spring web application context lifecycle support and configuration flexibility. The bean must implement `javax.servlet.Filter` and it must have the same name as that in the `filter-name` element.

There is a lifecycle issue to consider when hosting `Filters` in an IoC container instead of a servlet container. Specifically, which container should be responsible for calling the `Filter`'s "startup" and "shutdown" methods? It is noted that the order of initialization and destruction of a `Filter` can vary by servlet container, and this can cause problems if one `Filter` depends on configuration settings established by an earlier initialized `Filter`. The Spring IoC container on the other hand has more comprehensive lifecycle/IoC interfaces (such as `InitializingBean`, `DisposableBean`, `BeanNameAware`, `ApplicationContextAware` and many others) as well as a well-understood interface contract, predictable method invocation ordering, autowiring support, and even options to avoid implementing Spring interfaces (eg the `destroy-method` attribute in Spring XML). For this reason we recommend the use of Spring lifecycle services instead of servlet container lifecycle services wherever possible. Read the Javadoc for `DelegatingFilterProxy` for more information

Rather than using `DelegatingFilterProxy`, we strongly recommend that you use `FilterChainProxy` instead. Whilst `DelegatingFilterProxy` is a very useful class, the problem is that the number of lines of code required for `<filter>` and `<filter-mapping>` entries in `web.xml` explodes when using more than a few filters. To overcome this issue, Spring Security provides a `FilterChainProxy` class. It is wired using a `DelegatingFilterProxy` (just like in the example above), but the target class is `org.springframework.security.util.FilterChainProxy`. The filter chain is then declared in the application context, using code such as this:

```
<bean id="filterChainProxy" class="org.springframework.security.util.FilterChainProxy">
  <sec:filter-chain-map path-type="ant">
     <sec:filter-chain pattern="/webServices/**"
         filters="httpSessionContextIntegrationFilterWithASCFalse,basicProcessingFilter,exceptionTranslationFilt
     <sec:filter-chain pattern="/**"
         filters="httpSessionContextIntegrationFilterWithASCTrue,authenticationProcessingFilter,exceptionTransla
  </sec:filter-chain-map>
</bean>
```

You may notice similarities with the way `FilterSecurityInterceptor` is declared. Both regular expressions

and Ant Paths are supported, and the most specific URIs appear first. At runtime the `FilterChainProxy` will locate the first URI pattern that matches the current web request and the list of filter beans specified by the `filters` attribute will be applied to that request. The filters will be invoked in the order they are defined, so you have complete control over the filter chain which is applied to a particular URL.

You may have noticed we have declared two `HttpSessionContextIntegrationFilters` in the filter chain (`ASC` is short for `allowSessionCreation`, a property of `HttpSessionContextIntegrationFilter`). As web services will never present a `jsessionid` on future requests, creating `HttpSessions` for such user agents would be wasteful. If you had a high-volume application which required maximum scalability, we recommend you use the approach shown above. For smaller applications, using a single `HttpSessionContextIntegrationFilter` (with its default `allowSessionCreation` as `true`) would likely be sufficient.

In relation to lifecycle issues, the `FilterChainProxy` will always delegate `init(FilterConfig)` and `destroy()` methods through to the underlaying `Filters` if such methods are called against `FilterChainProxy` itself. In this case, `FilterChainProxy` guarantees to only initialize and destroy each `Filter` once, irrespective of how many times it is declared by the `FilterInvocationDefinitionSource`. You control the overall choice as to whether these methods are called or not via the `targetFilterLifecycle` initialization parameter of the `DelegatingFilterProxy` that proxies `DelegatingFilterProxy`. As discussed above, by default any servlet container lifecycle invocations are not delegated through to `DelegatingFilterProxy`.

You can use the attribute `filters = "none"` in the same way that you do when using namespace configuration to build the `FilterChainProxy`. This will omit the request pattern from the security filter chain entirely. Note that anything matching this path will then have no authentication or authorization services applied and will be freely accessible.

The order that filters are defined in `web.xml` is very important. Irrespective of which filters you are actually using, the order of the `<filter-mapping>`s should be as follows:

1. `ChannelProcessingFilter`, because it might need to redirect to a different protocol
2. `ConcurrentSessionFilter`, because it doesn't use any `SecurityContextHolder` functionality but needs to update the `SessionRegistry` to reflect ongoing requests from the principal
3. `HttpSessionContextIntegrationFilter`, so a `SecurityContext` can be setup in the `SecurityContextHolder` at the beginning of a web request, and any changes to the `SecurityContext` can be copied to the `HttpSession` when the web request ends (ready for use with the next web request)
4. Authentication processing mechanisms - `AuthenticationProcessingFilter`, `CasProcessingFilter`, `BasicProcessingFilter`, `HttpRequestIntegrationFilter`, `JbossIntegrationFilter` etc - so that the `SecurityContextHolder` can be modified to contain a valid `Authentication` request token
5. The `SecurityContextHolderAwareRequestFilter`, if you are using it to install a Spring Security aware `HttpServletRequestWrapper` into your servlet container
6. `RememberMeProcessingFilter`, so that if no earlier authentication processing mechanism updated the `SecurityContextHolder`, and the request presents a cookie that enables remember-me services to take place, a suitable remembered `Authentication` object will be put there
7. `AnonymousProcessingFilter`, so that if no earlier authentication processing mechanism updated the `SecurityContextHolder`, an anonymous `Authentication` object will be put there
8. `ExceptionTranslationFilter`, to catch any Spring Security exceptions so that either an HTTP error response can be returned or an appropriate `AuthenticationEntryPoint` can be launched
9. `FilterSecurityInterceptor`, to protect web URIs

All of the above filters use `DelegatingFilterProxy` or `FilterChainProxy`. It is recommended that a single `DelegatingFilterProxy` proxy through to a single `FilterChainProxy` for each application, with that `FilterChainProxy` defining all of Spring Security filters.

If you're using SiteMesh, ensure Spring Security filters execute before the SiteMesh filters are called. This enables the `SecurityContextHolder` to be populated in time for use by SiteMesh decorators

6.3. Tag Libraries

Spring Security comes bundled with several JSP tag libraries which provide a range of different services.

6.3.1. Configuration

All taglib classes are included in the core `spring-security-xx.jar` file, with the `security.tld` located in the JAR's `META-INF` directory. This means for JSP 1.2+ web containers you can simply include the JAR in the WAR's `WEB-INF/lib` directory and it will be available. If you're using a JSP 1.1 container, you'll need to declare the JSP taglib in your `web.xml file`, and include `security.tld` in the `WEB-INF/lib` directory. The following fragment is added to `web.xml`:

```
<taglib>
  <taglib-uri>http://www.springframework.org/security/tags</taglib-uri>
  <taglib-location>/WEB-INF/security.tld</taglib-location>
</taglib>
```

6.3.2. Usage

Now that you've configured the tag libraries, refer to the individual reference guide sections for details on how to use them. Note that when using the tags, you should include the taglib reference in your JSP:

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

Chapter 7. Channel Security

7.1. Overview

In addition to coordinating the authentication and authorization requirements of your application, Spring Security is also able to ensure unauthenticated web requests have certain properties. These properties may include being of a particular transport type, having a particular `HttpSession` attribute set and so on. The most common requirement is for your web requests to be received using a particular transport protocol, such as HTTPS.

An important issue in considering transport security is that of session hijacking. Your web container manages a `HttpSession` by reference to a `jsessionid` that is sent to user agents either via a cookie or URL rewriting. If the `jsessionid` is ever sent over HTTP, there is a possibility that session identifier can be intercepted and used to impersonate the user after they complete the authentication process. This is because most web containers maintain the same session identifier for a given user, even after they switch from HTTP to HTTPS pages.

If session hijacking is considered too significant a risk for your particular application, the only option is to use HTTPS for every request. This means the `jsessionid` is never sent across an insecure channel. You will need to ensure your `web.xml`-defined `<welcome-file>` points to an HTTPS location, and the application never directs the user to an HTTP location. Spring Security provides a solution to assist with the latter.

7.2. Configuration

Channel security is supported by the security namespace by means of the `requires-channel` attribute on the `<intercept-url>` element and this is the simplest (and recommended approach)

To confiure channel security explicitly, you would define the following the filter in your application context:

```
<bean id="channelProcessingFilter" class="org.springframework.security.securechannel.ChannelProcessingFilter">
  <property name="channelDecisionManager" ref="channelDecisionManager"/>
  <property name="filterInvocationDefinitionSource">
    <security:filter-invocation-definition-source path-type="regex">
      <security:intercept-url pattern="\A/secure/.*\Z" access="REQUIRES_SECURE_CHANNEL"/>
      <security:intercept-url pattern="\A/acegilogin.jsp.*\Z" access="REQUIRES_SECURE_CHANNEL"/>
      <security:intercept-url pattern="\A/j_spring_security_check.*\Z" access="REQUIRES_SECURE_CHANNEL"/>
      <security:intercept-url pattern="\A/.*\Z" access="ANY_CHANNEL"/>
    </security:filter-invocation-definition-source>
  </property>
</bean>

<bean id="channelDecisionManager" class="org.springframework.security.securechannel.ChannelDecisionManagerImpl">
  <property name="channelProcessors">
    <list>
    <ref bean="secureChannelProcessor"/>
    <ref bean="insecureChannelProcessor"/>
    </list>
  </property>
</bean>

<bean id="secureChannelProcessor" class="org.springframework.security.securechannel.SecureChannelProcessor"/>
<bean id="insecureChannelProcessor" class="org.springframework.security.securechannel.InsecureChannelProcessor"/
```

Like `FilterSecurityInterceptor`, Apache Ant style paths are also supported by the `ChannelProcessingFilter`.

The `ChannelProcessingFilter` operates by filtering all web requests and determining the configuration attributes that apply. It then delegates to the `ChannelDecisionManager`. The default implementation,

`ChannelDecisionManagerImpl`, should suffice in most cases. It simply delegates to the list of configured `ChannelProcessor` instances. The attribute `ANY_CHANNEL` can be used to override this behaviour and skip a particular URL. Otherwise, a `ChannelProcessor` will review the request, and if it is unhappy with the request (e.g. if it was received across the incorrect transport protocol), it will perform a redirect, throw an exception or take whatever other action is appropriate.

Included with Spring Security are two concrete `ChannelProcessor` implementations: `SecureChannelProcessor` ensures requests with a configuration attribute of `REQUIRES_SECURE_CHANNEL` are received over HTTPS, whilst `InsecureChannelProcessor` ensures requests with a configuration attribute of `REQUIRES_INSECURE_CHANNEL` are received over HTTP. Both implementations delegate to a `ChannelEntryPoint` if the required transport protocol is not used. The two `ChannelEntryPoint` implementations included with Spring Security simply redirect the request to HTTP and HTTPS as appropriate. Appropriate defaults are assigned to the `ChannelProcessor` implementations for the configuration attribute keywords they respond to and the `ChannelEntryPoint` they delegate to, although you have the ability to override these using the application context.

Note that the redirections are absolute (eg `http://www.company.com:8080/app/page`), not relative (eg `/app/page`). During testing it was discovered that Internet Explorer 6 Service Pack 1 has a bug whereby it does not respond correctly to a redirection instruction which also changes the port to use. Accordingly, absolute URLs are used in conjunction with bug detection logic in the `PortResolverImpl` that is wired up by default to many Spring Security beans. Please refer to the JavaDocs for `PortResolverImpl` for further details.

You should note that using a secure channel is recommended if usernames and passwords are to be kept secure during the login process. If you do decide to use `ChannelProcessingFilter` with form-based login, please ensure that your login page is set to `REQUIRES_SECURE_CHANNEL`, and that the `AuthenticationProcessingFilterEntryPoint.forceHttps` property is `true`.

7.3. Conclusion

Once configured, using the channel security filter is very easy. Simply request pages without regard to the protocol (ie HTTP or HTTPS) or port (eg 80, 8080, 443, 8443 etc). Obviously you'll still need a way of making the initial request (probably via the `web.xml` `<welcome-file>` or a well-known home page URL), but once this is done the filter will perform redirects as defined by your application context.

You can also add your own `ChannelProcessor` implementations to the `ChannelDecisionManagerImpl`. For example, you might set a `HttpSession` attribute when a human user is detected via a "enter the contents of this graphic" procedure. Your `ChannelProcessor` would respond to say `REQUIRES_HUMAN_USER` configuration attributes and redirect to an appropriate entry point to start the human user validation process if the `HttpSession` attribute is not currently set.

To decide whether a security check belongs in a `ChannelProcessor` or an `AccessDecisionVoter`, remember that the former is designed to handle unauthenticated requests, whilst the latter is designed to handle authenticated requests. The latter therefore has access to the granted authorities of the authenticated principal. In addition, problems detected by a `ChannelProcessor` will generally cause an HTTP/HTTPS redirection so its requirements can be met, whilst problems detected by an `AccessDecisionVoter` will ultimately result in an `AccessDeniedException` (depending on the governing `AccessDecisionManager`).

Part III. Authentication

We've already introduced Spring Security's authentication architecture in the Technical Overview chapter. In this part of the reference guide we will examine individual authentication mechanisms and their corresponding `AuthenticationProviders`. We'll also look at how to configure authentication more generally, including if you have several authentication approaches that need to be chained together.

With some exceptions, we will be discussing the full details of Spring Security bean configuration rather than the shorthand namespace syntax. You should review the introduction to using namespace configuration and the options it provides to see if they will meet your needs. As you come to use the framework more, and need to customize the internal behaviour, you will probably want to understand more about how the individual services are implemented, which classes to look at extending and so on. This part is more targeted at providing this kind of information. We'd recommend that you supplement the content by browsing the Javadoc and the source itself [1].

[1] Links to both Javadoc APIs and browsable source cross-reference are available from the project web site.

Chapter 8. Common Authentication Services

8.1. Mechanisms, Providers and Entry Points

To use Spring Security's authentication services, you'll usually need to configure a web filter, together with an `AuthenticationProvider` and `AuthenticationEntryPoint`. In this section we are going to explore an example application that needs to support both form-based authentication (so a nice HTML page is presented to a user for them to login) and BASIC authentication (so a web service or similar can access protected resources).

In the web.xml, this application will need a single Spring Security filter in order to use the FilterChainProxy. Nearly every Spring Security application will have such an entry, and it looks like this:

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

<filter-mapping>
  <filter-name>filterChainProxy</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

The above declarations will cause every web request to be passed through to the bean called `filterChainProxy` which will usually be an instance of Spring Security's `FilterChainProxy`. As explained in the filters section of this reference guide, the `FilterChainProxy` is a generally-useful class that enables web requests to be passed to different filters based on URL patterns. Those delegated filters are managed inside the application context, so they can benefit from dependency injection. Let's have a look at what the FilterChainProxy bean definition would look like inside your application context:

```
<bean id="filterChainProxy"
        class="org.springframework.security.util.FilterChainProxy">
  <security:filter-chain-map path-type="ant">
    <security:filter-chain pattern="/**" filters="httpSessionContextIntegrationFilter,logoutFilter,authenticatio
  </security:filter-chain-map>
</bean>
```

The `filter-chain-map` syntax from the security namespace allows you to define the mapping from URLs to filter chains, using a sequence of `filter-chain` child elements. Each of these defines a set of URLs using the `pattern` attribute and a chain of filters using the `filters` attribute.What's important to note at this stage is that a series of filters will be run - in the order specified by the declaration - and each of those filters are actually the `id` of another bean in the application context. So, in our case some extra beans will also appear in the application context, and they'll be named `httpSessionContextIntegrationFilter`, `logoutFilter` and so on. The order that the filters should appear is discussed in the filters section of the reference guide - although they are correct in the above example.

In our example we have the `AuthenticationProcessingFilter` and `BasicProcessingFilter` being used. These are the "authentication mechanisms" that respond to form-based authentication and BASIC HTTP header-based authentication respectively (we discussed the role of authentication mechanisms earlier in this reference guide). If you weren't using form or BASIC authentication, neither of these beans would be defined. You'd instead define filters applicable to your desired authentication environment, such as `DigestProcessingFilter` or `CasProcessingFilter`. Refer to the individual chapters of this part of the

reference guide to learn how to configure each of these authentication mechanisms.

Recall that `HttpSessionContextIntegrationFilter` keeps the contents of the `SecurityContext` between invocations inside an HTTP session. This means the authentication mechanisms are only used once, being when the principal initially tries to authenticate. The rest of the time the authentication mechanisms sit there and silently pass the request through to the next filter in the chain. That is a practical requirement due to the fact that few authentication approaches present credentials on each and every call (BASIC authentication being a notable exception), but what happens if a principal's account gets cancelled or disabled or otherwise changed (eg an increase or decrease in `GrantedAuthority[]`s) after the initial authentication step? Let's look at how that is handled now.

The major authorization provider for secure objects has previously been introduced as `AbstractSecurityInterceptor`. This class needs to have access to an `AuthenticationManager`. It also has configurable settings to indicate whether an `Authentication` object should be re-authenticated on each secure object invocation. By default it just accepts any `Authentication` inside the `SecurityContextHolder` is authenticated if `Authentication.isAuthenticated()` returns true. This is great for performance, but not ideal if you want to ensure up-to-the-moment authentication validity. For such cases you'll probably want to set the `AbstractSecurityInterceptor.alwaysReauthenticate` property to true.

You might be asking yourself, "what's this `AuthenticationManager`?". We haven't explored it before, but we have discussed the concept of an `AuthenticationProvider`. Quite simply, an `AuthenticationManager` is responsible for passing requests through a chain of AuthenticationProviders. It's a little like the filter chain we discussed earlier, although there are some differences. There is only one `AuthenticationManager` implementation shipped with Spring Security, so let's look at how it's configured for the example we're using in this chapter:

```
<bean id="authenticationManager"
        class="org.springframework.security.providers.ProviderManager">
<property name="providers">
<list>
  <ref local="daoAuthenticationProvider"/>
  <ref local="anonymousAuthenticationProvider"/>
  <ref local="rememberMeAuthenticationProvider"/>
</list>
</property>
</bean>
```

It's probably worth mentioning at this point that your authentication mechanisms (which are usually filters) are also injected with a reference to the `AuthenticationManager`. So both `AbstractSecurityInterceptor` as well as the authentication mechanisms will use the above `ProviderManager` to poll a list of `AuthenticationProviders`.

In our example we have three providers. They are tried in the order shown (which is implied by the use of a `List` instead of a `Set`), with each provider able to attempt authentication, or skip authentication by simply returning `null`. If all implementations return null, the `ProviderManager` will throw a suitable exception. If you're interested in learning more about chaining providers, please refer to the `ProviderManager` JavaDocs.

The providers to use will sometimes be interchangeable with the authentication mechanisms, whilst at other times they will depend on a specific authentication mechanism. For example, the `DaoAuthenticationProvider` just needs a string-based username and password. Various authentication mechanisms result in the collection of a string-based username and password, including (but not limited to) BASIC and form authentication. Equally, some authentication mechanisms create an authentication request object which can only be interpreted by a single type of `AuthenticationProvider`. An example of this one-to-one mapping would be JA-SIG CAS, which uses the notion of a service ticket which can therefore only be authenticated by `CasAuthenticationProvider`. A further example of a one-to-one mapping would be the LDAP authentication mechanism, which can only be processed an the `LdapAuthenticationProvider`. The specifics of such

relationships are detailed in the JavaDocs for each class, plus the authentication approach-specific chapters of this reference guide. You need not be terribly concerned about this implementation detail, because if you forget to register a suitable provider, you'll simply receive a `ProviderNotFoundException` when an attempt to authenticate is made.

After configuring the correct authentication mechanisms in the `FilterChainProxy`, and ensuring that a corresponding `AuthenticationProvider` is registered in the `ProviderManager`, your last step is to configure an `AuthenticationEntryPoint`. Recall that earlier we discussed the role of `ExceptionTranslationFilter`, which is used when HTTP-based requests should receive back an HTTP header or HTTP redirect in order to start authentication. Continuing on with our earlier example:

```
<bean id="exceptionTranslationFilter"
        class="org.springframework.security.ui.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint" ref="authenticationProcessingFilterEntryPoint"/>
  <property name="accessDeniedHandler">
    <bean class="org.springframework.security.ui.AccessDeniedHandlerImpl">
      <property name="errorPage" value="/accessDenied.jsp"/>
    </bean>
  </property>
</bean>

<bean id="authenticationProcessingFilterEntryPoint"
        class="org.springframework.security.ui.webapp.AuthenticationProcessingFilterEntryPoint">
  <property name="loginFormUrl" value="/login.jsp"/>
  <property name="forceHttps">< value="false"/>
</bean>
```

Notice that the `ExceptionTranslationFilter` requires two collaborators. The first, `AccessDeniedHandlerImpl`, uses a `RequestDispatcher` forward to display the specified access denied error page. We use a forward so that the `SecurityContextHolder` still contains details of the principal, which may be useful for display to the user (in old releases of Spring Security we relied upon the servlet container to handle a 403 error message, which lacked this useful contextual information). `AccessDeniedHandlerImpl` will also set the HTTP header to 403, which is the official error code to indicate access denied. In the case of the `AuthentionEntryPoint`, here we're setting what action we would like taken when an unauthenticated principal attempts to perform a protected operation. Because in our example we're going to be using form-based authentication, we specify `AuthenticationProcessinFilterEntryPoint` and the URL of the login page. Your application will usually only have one entry point, and most authentication approaches define their own specific `AuthenticationEntryPoint`. Details of which entry point to use for each authentication approach is discussed in the authentication approach-specific chapters of this reference guide.

8.2. UserDetails and Associated Types

As mentioned in the first part of the reference guide, most authentication providers take advantage of the `UserDetails` and `UserDetailsService` interfaces. The contract for this latter interface consists of a single method:

```
UserDetails loadUserByUsername(String username) throws UsernameNotFoundException, DataAccessException;
```

The returned `UserDetails` is an interface that provides getters that guarantee non-null provision of basic authentication information such as the username, password, granted authorities and whether the user is enabled or disabled. Most authentication providers will use a `UserDetailsService`, even if the username and password are not actually used as part of the authentication decision. Generally such providers will be using the returned

`UserDetails` object just for its `GrantedAuthority[]` information, because some other system (like LDAP or X509 or CAS etc) has undertaken the responsibility of actually validating the credentials.

A single concrete implementation of `UserDetails` is provided with Spring Security, being the `User` class. Spring Security users will need to decide when writing their `UserDetailsService` what concrete `UserDetails` class to return. In most cases `User` will be used directly or subclassed, although special circumstances (such as object relational mappers) may require users to write their own `UserDetails` implementation from scratch. This is not such an unusual situation, and users should not hesitate to simply return their normal domain object that represents a user of the system. This is especially common given that `UserDetails` is often used to store additional principal-related properties (such as their telephone number and email address), so that they can be easily used by web views.

Given `UserDetailsService` is so simple to implement, it should be easy for users to retrieve authentication information using a persistence strategy of their choice. Having said that, Spring Security does include a couple of useful base implementations, which we'll look at below.

8.2.1. In-Memory Authentication

Whilst it is easy to use create a custom `UserDetailsService` implementation that extracts information from a persistence engine of choice, many applications do not require such complexity. This is particularly true if you're undertaking a rapid prototype or just starting integrating Spring Security, when you don't really want to spend time configuring databases or writing `UserDetailsService` implementations. For this sort of situation, a simple option is to use the `user-service` element from the security namespace:

```
<user-service id="userDetailsService">
  <user name="jimi" password="jimispassword" authorities="ROLE_USER, ROLE_ADMIN" />
  <user name="bob" password="bobspassword" authorities="ROLE_USER" />
</user-service>
```

This also suppots the use of an external properties file:

```
<user-service id="userDetailsService" properties="users.properties"/>
```

The properties file should contain entries in the form

```
username=password,grantedAuthority[,grantedAuthority][,enabled|disabled]
```

For example

```
jimi=jimispassword,ROLE_USER,ROLE_ADMIN,enabled
bob=bobspassword,ROLE_USER,enabled
```

8.2.2. JDBC Authentication

Spring Security also includes a `UserDetailsService` that can obtain authentication information from a JDBC data source. Internally Spring JDBC is used, so it avoids the complexity of a fully-featured object relational mapper (ORM) just to store user details. If your application does use an ORM tool, you might prefer to write a custom `UserDetailsService` to reuse the mapping files you've probably already created. Returning to `JdbcDaoImpl`, an example configuration is shown below:

```
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>

<bean id="userDetailsService" class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

You can use different relational database management systems by modifying the `DriverManagerDataSource` shown above. You can also use a global data source obtained from JNDI, as per normal Spring options.

8.2.2.1. Default User Database Schema

Irrespective of the database you are using and how a `DataSource` is obtained, a standard schema must be in place. The DDL for an HSQL database instance would be:

```
  CREATE TABLE users (
  username VARCHAR(50) NOT NULL PRIMARY KEY,
  password VARCHAR(50) NOT NULL,
  enabled BIT NOT NULL
  );

  CREATE TABLE authorities (
  username VARCHAR(50) NOT NULL,
  authority VARCHAR(50) NOT NULL
  );

  ALTER TABLE authorities ADD CONSTRAINT fk_authorities_users foreign key (username) REFERENCES users(username);
```

If the default schema is unsuitable for your needs, `JdbcDaoImpl` provides properties that allow customisation of the SQL statements. Please refer to the JavaDocs for details, but note that the class is not intended for complex custom subclasses. If you have a complex schema or would like a custom `UserDetails` implementation returned, you'd be better off writing your own `UserDetailsService`. The base implementation provided with Spring Security is intended for typical situations, rather than catering for all possible requirements.

8.3. Concurrent Session Handling

Spring Security is able to prevent a principal from concurrently authenticating to the same application more than a specified number of times. Many ISVs take advantage of this to enforce licensing, whilst network administrators like this feature because it helps prevent people from sharing login names. You can, for example, stop user "Batman" from logging onto the web application from two different sessions.

To use concurrent session support, you'll need to add the following to `web.xml`:

```
<listener>
    <listener-class>org.springframework.security.ui.session.HttpSessionEventPublisher</listener-class>
</listener>
```

In addition, you will need to add the `org.springframework.security.concurrent.ConcurrentSessionFilter` to your `FilterChainProxy`. The

`ConcurrentSessionFilter` requires two properties, `sessionRegistry`, which generally points to an instance of `SessionRegistryImpl`, and `expiredUrl`, which points to the page to display when a session has expired.

The `web.xml` `HttpSessionEventPublisher` causes an `ApplicationEvent` to be published to the Spring `ApplicationContext` every time a `HttpSession` commences or terminates. This is critical, as it allows the `SessionRegistryImpl` to be notified when a session ends.

You will also need to wire up the `ConcurrentSessionControllerImpl` and refer to it from your `ProviderManager` bean:

```
<bean id="authenticationManager"
    class="org.springframework.security.providers.ProviderManager">
  <property name="providers">
    <!-- your providers go here -->
  </property>
  <property name="sessionController" ref="concurrentSessionController"/>
</bean>

<bean id="concurrentSessionController"
    class="org.springframework.security.concurrent.ConcurrentSessionControllerImpl">
  <property name="maximumSessions" value="1"/>
  <property name="sessionRegistry">
    <bean class="org.springframework.security.concurrent.SessionRegistryImpl"/>
  <property>
</bean>
```

8.4. Authentication Tag Libraries

`AuthenticationTag` is used to simply output a property of the current `Authentication` object to the web page.

The following JSP fragment illustrates how to use the `AuthenticationTag`:

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

This tag would cause the principal's name to be output. Here we are assuming the `Authentication.getPrincipal()` is a `UserDetails` object, which is generally the case when using one of Spring Security's stadard `AuthenticationProvider` implementations.

Chapter 9. DAO Authentication Provider

9.1. Overview

Spring Security includes a production-quality `AuthenticationProvider` implementation called `DaoAuthenticationProvider`. This authentication provider is compatible with all of the authentication mechanisms that generate a `UsernamePasswordAuthenticationToken`, and is probably the most commonly used provider in the framework. Like most of the other authentication providers, the DaoAuthenticationProvider leverages a UserDetailsService in order to lookup the username, password and GrantedAuthority[]s. Unlike most of the other authentication providers that leverage UserDetailsService, this authentication provider actually requires the password to be presented, and the provider will actually evaluate the validity or otherwise of the password presented in an authentication request object.

9.2. Configuration

Aside from adding DaoAuthenticationProvider to your ProviderManager list (as discussed at the start of this part of the reference guide), and ensuring a suitable authentication mechanism is configured to present a UsernamePasswordAuthenticationToken, the configuration of the provider itself is rather simple:

```
<bean id="daoAuthenticationProvider"
    class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="inMemoryDaoImpl"/>
  <property name="saltSource" ref bean="saltSource"/>
  <property name="passwordEncoder" ref="passwordEncoder"/>
</bean>
```

The `PasswordEncoder` and `SaltSource` are optional. A `PasswordEncoder` provides encoding and decoding of passwords presented in the `UserDetails` object that is returned from the configured `UserDetailsService`. A `SaltSource` enables the passwords to be populated with a "salt", which enhances the security of the passwords in the authentication repository. `PasswordEncoder` implementations are provided with Spring Security covering MD5, SHA and cleartext encodings. Two `SaltSource` implementations are also provided: `SystemWideSaltSource` which encodes all passwords with the same salt, and `ReflectionSaltSource`, which inspects a given property of the returned `UserDetails` object to obtain the salt. Please refer to the JavaDocs for further details on these optional features.

In addition to the properties above, the `DaoAuthenticationProvider` supports optional caching of `UserDetails` objects. The `UserCache` interface enables the `DaoAuthenticationProvider` to place a `UserDetails` object into the cache, and retrieve it from the cache upon subsequent authentication attempts for the same username. By default the `DaoAuthenticationProvider` uses the `NullUserCache`, which performs no caching. A usable caching implementation is also provided, `EhCacheBasedUserCache`, which is configured as follows:

```
<bean id="daoAuthenticationProvider"
    class="org.springframework.security.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="userDetailsService"/>
  <property name="userCache" ref="userCache"/>
</bean>

<bean id="cacheManager" class="org.springframework.cache.ehcache.EhCacheManagerFactoryBean">
```

```
  <property name="configLocation" value="classpath:/ehcache-failsafe.xml"/>
</bean>

<bean id="userCacheBackend" class="org.springframework.cache.ehcache.EhCacheFactoryBean">
  <property name="cacheManager" ref="cacheManager"/>
  <property name="cacheName" value="userCache"/>
</bean>

<bean id="userCache" class="org.springframework.security.providers.dao.cache.EhCacheBasedUserCache">
  <property name="cache" ref="userCacheBackend"/>
</bean>
```

All Spring Security EH-CACHE implementations (including `EhCacheBasedUserCache`) require an EH-CACHE `Cache` object. The `Cache` object can be obtained from wherever you like, although we recommend you use Spring's factory classes as shown in the above configuration. If using Spring's factory classes, please refer to the Spring documentation for further details on how to optimise the cache storage location, memory usage, eviction policies, timeouts etc.

Note

In the majority of cases, where your application is a stateful web application, you don't need to use a cache as the user's authentication information will be stored in the `HttpSession`.

A design decision was made not to support account locking in the `DaoAuthenticationProvider`, as doing so would have increased the complexity of the `UserDetailsService` interface. For instance, a method would be required to increase the count of unsuccessful authentication attempts. Such functionality could be easily provided by leveraging the application event publishing features discussed below.

`DaoAuthenticationProvider` returns an `Authentication` object which in turn has its `principal` property set. The principal will be either a `String` (which is essentially the username) or a `UserDetails` object (which was looked up from the `UserDetailsService`). By default the `UserDetails` is returned, as this enables applications to add extra properties potentially of use in applications, such as the user's full name, email address etc. If using container adapters, or if your applications were written to operate with `String`s (as was the case for releases prior to Spring Security 0.6), you should set the `DaoAuthenticationProvider.forcePrincipalAsString` property to `true` in your application context

Chapter 10. LDAP Authentication

10.1. Overview

LDAP is often used by organizations as a central repository for user information and as an authentication service. It can also be used to store the role information for application users.

There are many different scenarios for how an LDAP server may be configured so Spring Security's LDAP provider is fully configurable. It uses separate strategy interfaces for authentication and role retrieval and provides default implementations which can be configured to handle a wide range of situations.

You should be familiar with LDAP before trying to use it with Spring Security. The following link provides a good introduction to the concepts involved and a guide to setting up a directory using the free LDAP server OpenLDAP: `http://www.zytrax.com/books/ldap/`. Some familiarity with the JNDI APIs used to access LDAP from Java may also be useful. We don't use any third-party LDAP libraries (Mozilla, JLDAP etc.) in the LDAP provider, but extensive use is made of Spring LDAP, so some familiarity with that project may be useful if you plan on adding your own customizations.

10.2. Using LDAP with Spring Security

LDAP authentication in Spring Security can be roughly divided into the following stages.

1. Obtaining the unique LDAP “Distinguished Name”, or DN, from the login name. This will often mean performing a search in the directory, unless the exact mapping of usernames to DNs is known in advance.

2. Authenticating the user, either by binding as that user or by performing a remote “compare” operation of the user's password against the password attribute in the directory entry for the DN.

3. Loading the list of authorities for the user.

The exception is when the LDAP directory is just being used to retrieve user information and authenticate against it locally. This may not be possible as directories are often set up with limited read access for attributes such as user passwords.

We will look at some configuration scenarios below. For full information on available configuration options, please consult the security namespace schema (information from which should be available in your XML editor).

10.3. Configuring an LDAP Server

The first thing you need to do is configure the server against which authentication should take place. This is done using the `<ldap-server>` element from the security namespace. This can be configured to point at an external LDAP server, using the `url` attribute:

```
<ldap-server url="ldap://springframework.org:389/dc=springframework,dc=org" />
```

10.3.1. Using an Embedded Test Server

The `<ldap-server>` element can also be used to create an embedded server, which can be very useful for testing and demonstrations. In this case you use it without the `url` attribute:

```
<ldap-server root="dc=springframework,dc=org"/>
```

Here we've specified that the root DIT of the directory should be “dc=springframework,dc=org”, which is the default. Used this way, the namespace parser will create an embedded Apache Directory server and scan the classpath for any LDIF files, which it will attempt to load into the server. You can customize this behaviour using the `ldif` attribute, which defines an LDIF resource to be loaded:

```
<ldap-server ldif="classpath:users.ldif" />
```

This makes it a lot easier to get up and running with LDAP, since it can be inconvenient to work all the time with an external server. It also insulates the user from the complex bean configuration needed to wire up an Apache Directory server. Using plain Spring Beans the configuration would be much more cluttered. You must have the necessary Apache Directory dependency jars available for your application to use. These can be obtained from the LDAP sample application.

10.3.2. Using Bind Authentication

This is the most common LDAP authentication scenario.

```
<ldap-authentication-provider user-dn-pattern="uid={0},ou=people"/>
```

This simple example would obtain the DN for the user by substituting the user login name in the supplied pattern and attempting to bind as that user with the login password. This is OK if all your users are stored under a single node in the directory. If instead you wished to configure an LDAP search filter to locate the user, you could use the following:

```
<ldap-authentication-provider user-search-filter="(uid={0})" user-search-base="ou=people"/>
```

If used with the server definition above, this would perform a search under the DN `ou=people,dc=springframework,dc=org` using the value of the `user-search-filter` attribute as a filter. Again the user login name is substituted for the parameter in the filter name. If `user-search-base` isn't supplied, the search will be performed from the root.

10.3.3. Loading Authorities

How authorities are loaded from groups in the LDAP directory is controlled by the following attributes.

- `group-search-base`. Defines the part of the directory tree under which group searches should be performed.
- `group-role-attribute`. The attribute which contains the name of the authority defined by the group entry. Defaults to `cn`

- `group-search-filter`. The filter which is used to search for group membership. The default is `uniqueMember={0}`, corresponding to the `groupOfUniqueMembers` LDAP class. In this case, the substituted parameter is the full distinguished name of the user. The parameter `{1}` can be used if you want to filter on the login name.

So if we used the following configuration

```
<ldap-authentication-provider user-dn-pattern="uid={0},ou=people" group-search-base="ou=groups" />
```

and authenticated successfully as user “ben”, the subsequent loading of authorities would perform a search under the directory entry `ou=groups,dc=springframework,dc=org`, looking for entries which contain the attribute `uniqueMember` with value `uid=ben,ou=people,dc=springframework,dc=org`. By default the authority names will have the prefix `ROLE_` prepended. You can change this using the `role-prefix` attribute. If you don't want any prefix, use `role-prefix="none"`. For more information on loading authorities, see the Javadoc for the `DefaultLdapAuthoritiesPopulator` class.

10.4. Implementation Classes

The namespace configuration options we've used above are simple to use and much more concise than using Spring beans explicitly. There are situations when you may need to know how to configure Spring Security LDAP directly in your application context. You may wish to customize the behaviour of some of the classes, for example. If you're happy using namespace configuration then you can skip this section and the next one.

The main LDAP provider class is `org.springframework.security.providers.ldap.LdapAuthenticationProvider`. This bean doesn't actually do much itself but delegates the work to two other beans, an `LdapAuthenticator` and an `LdapAuthoritiesPopulator` which are responsible for authenticating the user and retrieving the user's set of `GrantedAuthority`s respectively.

10.4.1. LdapAuthenticator Implementations

The authenticator is also responsible for retrieving any required user attributes. This is because the permissions on the attributes may depend on the type of authentication being used. For example, if binding as the user, it may be necessary to read them with the user's own permissions.

There are currently two authentication strategies supplied with Spring Security:

- Authentication directly to the LDAP server ("bind" authentication).
- Password comparison, where the password supplied by the user is compared with the one stored in the repository. This can either be done by retrieving the value of the password attribute and checking it locally or by performing an LDAP "compare" operation, where the supplied password is passed to the server for comparison and the real password value is never retrieved.

10.4.1.1. Common Functionality

Before it is possible to authenticate a user (by either strategy), the distinguished name (DN) has to be obtained from the login name supplied to the application. This can be done either by simple pattern-matching (by setting the setUserDnPatterns array property) or by setting the userSearch property. For the DN pattern-matching approach, a standard Java pattern format is used, and the login name will be substituted for the parameter *`{0}`*. The pattern should be relative to the DN that the configured `SpringSecurityContextSource` will bind to (see

the section on connecting to the LDAP server for more information on this). For example, if you are using an LDAP server with the URL `ldap://monkeymachine.co.uk/dc=springframework,dc=org`, and have a pattern `uid={0},ou=greatapes`, then a login name of "gorilla" will map to a DN `uid=gorilla,ou=greatapes,dc=springframework,dc=org`. Each configured DN pattern will be tried in turn until a match is found. For information on using a search, see the section on search objects below. A combination of the two approaches can also be used - the patterns will be checked first and if no matching DN is found, the search will be used.

10.4.1.2. BindAuthenticator

The class `org.springframework.security.providers.ldap.authenticator.BindAuthenticator` implements the bind authentication strategy. It simply attempts to bind as the user.

10.4.1.3. PasswordComparisonAuthenticator

The class `org.springframework.security.providers.ldap.authenticator.PasswordComparisonAuthenticator` implements the password comparison authentication strategy.

10.4.1.4. Active Directory Authentication

In addition to standard LDAP authentication (binding with a DN), Active Directory has its own non-standard syntax for user authentication.

10.4.2. Connecting to the LDAP Server

The beans discussed above have to be able to connect to the server. They both have to be supplied with a `SpringSecurityContextSource` which is an extension of Spring LDAP's `ContextSource`. Unless you have special requirements, you will usually configure a `DefaultSpringSecurityContextSource` bean, which can be configured with the URL of your LDAP server and optionally with the username and password of a "manager" user which will be used by default when binding to the server (instead of binding anonymously). For more information read the Javadoc for this class and for Spring LDAP's `AbstractContextSource`.

10.4.3. LDAP Search Objects

Often more a more complicated strategy than simple DN-matching is required to locate a user entry in the directory. This can be encapsulated in an `LdapUserSearch` instance which can be supplied to the authenticator implementations, for example, to allow them to locate a user. The supplied implementation is `FilterBasedLdapUserSearch`.

10.4.3.1. `FilterBasedLdapUserSearch`

This bean uses an LDAP filter to match the user object in the directory. The process is explained in the Javadoc for the corresponding search method on the JDK DirContext class. As explained there, the search filter can be supplied with parameters. For this class, the only valid parameter is *`{0}`* which will be replaced with the user's login name.

10.4.4. LdapAuthoritiesPopulator

After authenticating the user successfully, the `LdapAuthenticationProvider` will attempt to load a set of authorities for the user by calling the configured `LdapAuthoritiesPopulator` bean. The

`DefaultLdapAuthoritiesPopulator` is an implementation which will load the authorities by searching the directory for groups of which the user is a member (typically these will be `groupOfNames` or `groupOfUniqueNames` entries in the directory). Consult the Javadoc for this class for more details on how it works.

10.4.5. Spring Bean Configuration

A typical configuration, using some of the beans we've discussed here, might look like this:

```
<bean id="contextSource"
        class="org.springframework.security.ldap.DefaultSpringSecurityContextSource">
  <constructor-arg value="ldap://monkeymachine:389/dc=springframework,dc=org"/>
  <property name="userDn" value="cn=manager,dc=springframework,dc=org"/>
  <property name="password" value="password"/>
</bean>

<bean id="ldapAuthProvider"
        class="org.springframework.security.providers.ldap.LdapAuthenticationProvider">
  <constructor-arg>
    <bean class="org.springframework.security.providers.ldap.authenticator.BindAuthenticator">
      <constructor-arg ref="contextSource"/>
      <property name="userDnPatterns">
        <list><value>uid={0},ou=people</value></list>
      </property>
    </bean>
  </constructor-arg>
  <constructor-arg>
    <bean class="org.springframework.security.ldap.populator.DefaultLdapAuthoritiesPopulator">
      <constructor-arg ref="contextSource"/>
      <constructor-arg value="ou=groups"/>
      <property name="groupRoleAttribute" value="ou"/>
    </bean>
  </constructor-arg>
</bean>
```

This would set up the provider to access an LDAP server with URL `ldap://monkeymachine:389/dc=springframework,dc=org`. Authentication will be performed by attempting to bind with the DN `uid=<user-login-name>,ou=people,dc=springframework,dc=org`. After successful authentication, roles will be assigned to the user by searching under the DN `ou=groups,dc=springframework,dc=org` with the default filter `(member=<user's-DN>)`. The role name will be taken from the “ou” attribute of each match.

To configure a user search object, which uses the filter `(uid=<user-login-name>)` for use instead of the DN-pattern (or in addition to it), you would configure the following bean

```
<bean id="userSearch"
    class="org.springframework.security.ldap.search.FilterBasedLdapUserSearch">
  <constructor-arg index="0" value=""/>
  <constructor-arg index="1" value="(uid={0})"/>
  <constructor-arg index="2" ref="contextSource" />
</bean>
```

and use it by setting the `BindAuthenticator` bean's userSearch property. The authenticator would then call the search object to obtain the correct user's DN before attempting to bind as this user.

10.4.6. LDAP Attributes and Customized UserDetails

The net result of an authentication using `LdapAuthenticationProvider` is the same as a normal Spring Security authentication using the standard `UserDetailsService` interface. A `UserDetails` object is created and stored in the returned `Authentication` object. As with using a `UserDetailsService`, a common requirement is

to be able to customize this implementation and add extra properties. When using LDAP, these will normally be attributes from the user entry. The creation of the `UserDetails` object is controlled by the provider's `UserDetailsContextMapper` strategy, which is responsible for mapping user objects to and from LDAP context data:

```
public interface UserDetailsContextMapper {
    UserDetails mapUserFromContext(DirContextOperations ctx, String username, GrantedAuthority[] authority);

    void mapUserToContext(UserDetails user, DirContextAdapter ctx);
}
```

Only the first method is relevant for authentication. If you provide an implementation of this interface, you can control exactly how the UserDetails object is created. The first parameter is an instance of Spring LDAP's `DirContextOperations` which gives you access to the LDAP attributes which were loaded. The `username` parameter is the name used to authenticate and the final parameter is the list of authorities loaded for the user.

The way the context data is loaded varies slightly depending on the type of authentication you are using. With the `BindAuthenticatior`, the context returned from the bind operation will be used to read the attributes, otherwise the data will be read using the standard context obtained from the configured `ContextSource` (when a search is configured to locate the user, this will be the data returned by the search object).

Chapter 11. Form Authentication Mechanism

11.1. Overview

HTTP Form Authentication involves using the `AuthenticationProcessingFilter` to process a login form. This is the most common way for an application to authenticate end users. Form-based authentication is entirely compatible with the DAO and JAAS authentication providers.

11.2. Configuration

The login form simply contains `j_username` and `j_password` input fields, and posts to a URL that is monitored by the filter (by default `/j_spring_security_check`). You should add an `AuthenticationProcessingFilter` to your application context:

```
<bean id="authenticationProcessingFilter"
    class="org.springframework.security.ui.webapp.AuthenticationProcessingFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="authenticationFailureUrl" value="/login.jsp?login_error=1"/>
  <property name="defaultTargetUrl" value="/"/>
  <property name="filterProcessesUrl" value="/j_spring_security_check"/>
</bean>
```

The configured `AuthenticationManager` processes each authentication request. If authentication fails, the browser will be redirected to the `authenticationFailureUrl`. The `AuthenticationException` will be placed into the `HttpSession` attribute indicated by `AbstractProcessingFilter.SPRING_SECURITY_LAST_EXCEPTION_KEY`, enabling a reason to be provided to the user on the error page.

If authentication is successful, the resulting `Authentication` object will be placed into the `SecurityContextHolder`.

Once the `SecurityContextHolder` has been updated, the browser will need to be redirected to the target URL which is usually indicated by the `HttpSession` attribute stored under `AbstractProcessingFilter.SPRING_SECURITY_TARGET_URL_KEY`. This attribute is automatically set by the `ExceptionTranslationFilter` when an `AuthenticationException` occurs, so that after login is completed the user can return to what they were originally trying to access. If for some reason the `HttpSession` does not indicate the target URL, the browser will be redirected to the `defaultTargetUrl` property.

Chapter 12. BASIC Authentication Mechanism

12.1. Overview

Spring Security provides a `BasicProcessingFilter` which is capable of processing basic authentication credentials presented in HTTP headers. This can be used for authenticating calls made by Spring remoting protocols (such as Hessian and Burlap), as well as normal user agents (such as Internet Explorer and Navigator). The standard governing HTTP Basic Authentication is defined by RFC 1945, Section 11, and the `BasicProcessingFilter` conforms with this RFC. Basic Authentication is an attractive approach to authentication, because it is very widely deployed in user agents and implementation is extremely simple (it's just a Base64 encoding of the username:password, specified in an HTTP header).

12.2. Configuration

To implement HTTP Basic Authentication, it is necessary to define `BasicProcessingFilter` in the filter chain. The application context will need to define the `BasicProcessingFilter` and its required collaborator:

```
<bean id="basicProcessingFilter" class="org.springframework.security.ui.basicauth.BasicProcessingFilter"
<property name="authenticationManager"><ref bean="authenticationManager"/></property>
<property name="authenticationEntryPoint"><ref bean="authenticationEntryPoint"/></property>
</bean>

<bean id="authenticationEntryPoint"
class="org.springframework.security.ui.basicauth.BasicProcessingFilterEntryPoint">
<property name="realmName"><value>Name Of Your Realm</value></property>
</bean>
```

The configured `AuthenticationManager` processes each authentication request. If authentication fails, the configured `AuthenticationEntryPoint` will be used to retry the authentication process. Usually you will use the `BasicProcessingFilterEntryPoint`, which returns a 401 response with a suitable header to retry HTTP Basic authentication. If authentication is successful, the resulting `Authentication` object will be placed into the `SecurityContextHolder`.

If the authentication event was successful, or authentication was not attempted because the HTTP header did not contain a supported authentication request, the filter chain will continue as normal. The only time the filter chain will be interrupted is if authentication fails and the `AuthenticationEntryPoint` is called, as discussed in the previous paragraph

Chapter 13. Digest Authentication

13.1. Overview

Spring Security provides a `DigestProcessingFilter` which is capable of processing digest authentication credentials presented in HTTP headers. Digest Authentication attempts to solve many of the weaknesses of Basic authentication, specifically by ensuring credentials are never sent in clear text across the wire. Many user agents support Digest Authentication, including FireFox and Internet Explorer. The standard governing HTTP Digest Authentication is defined by RFC 2617, which updates an earlier version of the Digest Authentication standard prescribed by RFC 2069. Most user agents implement RFC 2617. Spring Security `DigestProcessingFilter` is compatible with the "`auth`" quality of protection (`qop`) prescribed by RFC 2617, which also provides backward compatibility with RFC 2069. Digest Authentication is a highly attractive option if you need to use unencrypted HTTP (ie no TLS/HTTPS) and wish to maximise security of the authentication process. Indeed Digest Authentication is a mandatory requirement for the WebDAV protocol, as noted by RFC 2518 Section 17.1, so we should expect to see it increasingly deployed and replacing Basic Authentication.

Digest Authentication is definitely the most secure choice between Form Authentication, Basic Authentication and Digest Authentication, although extra security also means more complex user agent implementations. Central to Digest Authentication is a "nonce". This is a value the server generates. Spring Security's nonce adopts the following format:

```
base64(expirationTime + ":" + md5Hex(expirationTime + ":" + key))

expirationTime:   The date and time when the nonce expires, expressed in milliseconds
key:              A private key to prevent modification of the nonce token
```

The `DigestProcessingFilterEntryPoint` has a property specifying the `key` used for generating the nonce tokens, along with a `nonceValiditySeconds` property for determining the expiration time (default 300, which equals five minutes). Whist ever the nonce is valid, the digest is computed by concatenating various strings including the username, password, nonce, URI being requested, a client-generated nonce (merely a random value which the user agent generates each request), the realm name etc, then performing an MD5 hash. Both the server and user agent perform this digest computation, resulting in different hash codes if they disagree on an included value (eg password). In Spring Security implementation, if the server-generated nonce has merely expired (but the digest was otherwise valid), the `DigestProcessingFilterEntryPoint` will send a `"stale=true"` header. This tells the user agent there is no need to disturb the user (as the password and username etc is correct), but simply to try again using a new nonce.

An appropriate value for `DigestProcessingFilterEntryPoint`'s `nonceValiditySeconds` parameter will depend on your application. Extremely secure applications should note that an intercepted authentication header can be used to impersonate the principal until the `expirationTime` contained in the nonce is reached. This is the key principle when selecting an appropriate setting, but it would be unusual for immensely secure applications to not be running over TLS/HTTPS in the first instance.

Because of the more complex implementation of Digest Authentication, there are often user agent issues. For example, Internet Explorer fails to present an "`opaque`" token on subsequent requests in the same session. Spring Security filters therefore encapsulate all state information into the "`nonce`" token instead. In our testing, Spring Security implementation works reliably with FireFox and Internet Explorer, correctly handling nonce timeouts etc.

13.2. Configuration

Now that we've reviewed the theory, let's see how to use it. To implement HTTP Digest Authentication, it is necessary to define `DigestProcessingFilter` in the fitler chain. The application context will need to define the `DigestProcessingFilter` and its required collaborators:

```
<bean id="digestProcessingFilter"
    class="org.springframework.security.ui.digestauth.DigestProcessingFilter">
  <property name="userDetailsService" ref="jdbcDaoImpl"/>
  <property name="authenticationEntryPoint" ref="digestProcessingFilterEntryPoint"/>
  <property name="userCache" ref="userCache"/>
</bean>

<bean id="digestProcessingFilterEntryPoint"
    class="org.springframework.security.ui.digestauth.DigestProcessingFilterEntryPoint">
  <property name="realmName" value="Contacts Realm via Digest Authentication"/>
  <property name="key" value="acegi"/>
  <property name="nonceValiditySeconds" value="10"/>
</bean>
```

The configured `UserDetailsService` is needed because `DigestProcessingFilter` must have direct access to the clear text password of a user. Digest Authentication will NOT work if you are using encoded passwords in your DAO. The DAO collaborator, along with the `UserCache`, are typically shared directly with a `DaoAuthenticationProvider`. The `authenticationEntryPoint` property must be `DigestProcessingFilterEntryPoint`, so that `DigestProcessingFilter` can obtain the correct `realmName` and `key` for digest calculations.

Like `BasicAuthenticationFilter`, if authentication is successful an `Authentication` request token will be placed into the `SecurityContextHolder`. If the authentication event was successful, or authentication was not attempted because the HTTP header did not contain a Digest Authentication request, the filter chain will continue as normal. The only time the filter chain will be interrupted is if authentication fails and the `AuthenticationEntryPoint` is called, as discussed in the previous paragraph.

Digest Authentication's RFC offers a range of additional features to further increase security. For example, the nonce can be changed on every request. Despite this, Spring Security implementation was designed to minimise the complexity of the implementation (and the doubtless user agent incompatibilities that would emerge), and avoid needing to store server-side state. You are invited to review RFC 2617 if you wish to explore these features in more detail. As far as we are aware, Spring Security's implementation does comply with the minimum standards of this RFC.

Chapter 14. Remember-Me Authentication

14.1. Overview

Remember-me or persistent-login authentication refers to web sites being able to remember the identity of a principal between sessions. This is typically accomplished by sending a cookie to the browser, with the cookie being detected during future sessions and causing automated login to take place. Spring Security provides the necessary hooks for these operations to take place, and has two concrete remember-me implementations. One uses hashing to preserve the security of cookie-based tokens and the other uses a database or other persistent storage mechanism to store the generated tokens.

Note that both implemementations require a `UserDetailsService`. If you are using an authentication provider which doesn't use a `UserDetailsService` (for example, the LDAP provider) then it won't work unless you also have a `UserDetailsService` bean in your application context.

14.2. Simple Hash-Based Token Approach

This approach uses hashing to achieve a useful remember-me strategy. In essence a cookie is sent to the browser upon successful interactive authentication, with the cookie being composed as follows:

```
base64(username + ":" + expirationTime + ":" + md5Hex(username + ":" + expirationTime + ":" password + ":" +

username:          As identifiable to the UserDetailsService
password:          That matches the one in the retrieved UserDetails
expirationTime:    The date and time when the remember-me token expires, expressed in milliseconds
key:               A private key to prevent modification of the remember-me token
```

As such the remember-me token is valid only for the period specified, and provided that the username, password and key does not change. Notably, this has a potential security issue in that a captured remember-me token will be usable from any user agent until such time as the token expires. This is the same issue as with digest authentication. If a principal is aware a token has been captured, they can easily change their password and immediately invalidate all remember-me tokens on issue. If more significant security is needed you should use the approach described in the next section. Alternatively remember-me services should simply not be used at all.

If you are familiar with the topics discussed in the chapter on namespace configuration, you can enable remember-me authentication just by adding the `<remember-me>` element:

```
<http>
  ...
  <remember-me key="myAppKey"/>
</http>
```

It is automatically enabled for you if you are using the auto-config setting. The `UserDetailsService` will normally be selected automatically. If you have more than one in your application context, you need to specify which one should be used with the `user-service-ref` attribute, where the value is the name of your `UserDetailsService` bean.

14.3. Persistent Token Approach

This approach is based on the article http://jaspan.com/improved_persistent_login_cookie_best_practice with some minor modifications [1]. To use the this approach with namespace configuration, you would supply a datasource reference:

```
<http>
  ...
  <remember-me data-source-ref="someDataSource"/>
</http>
```

The database should contain a `persistent_logins` table, created using the following SQL (or equivalent):

```
  create table persistent_logins (username varchar(64) not null, series varchar(64) primary key, token varchar
```

14.4. Remember-Me Interfaces and Implementations

Remember-me authentication is not used with basic authentication, given it is often not used with `HttpSessions`. Remember-me is used with `AuthenticationProcessingFilter`, and is implemented via hooks in the `AbstractProcessingFilter` superclass. The hooks will invoke a concrete `RememberMeServices` at the appropriate times. The interface looks like this:

```
Authentication autoLogin(HttpServletRequest request, HttpServletResponse response);
void loginFail(HttpServletRequest request, HttpServletResponse response);
void loginSuccess(HttpServletRequest request, HttpServletResponse response, Authentication successfulAuthentic
```

Please refer to the JavaDocs for a fuller discussion on what the methods do, although note at this stage that `AbstractProcessingFilter` only calls the `loginFail()` and `loginSuccess()` methods. The `autoLogin()` method is called by `RememberMeProcessingFilter` whenever the `SecurityContextHolder` does not contain an `Authentication`. This interface therefore provides the underlying remember-me implementation with sufficient notification of authentication-related events, and delegates to the implementation whenever a candidate web request might contain a cookie and wish to be remembered. This design allows any number of remember-me implementation strategies. We've seen above that Spring Security provides two implementations. We'll look at thes in turn.

14.4.1. TokenBasedRememberMeServices

This implementation supports the simpler approach described in Section 14.2, “Simple Hash-Based Token Approach”. `TokenBasedRememberMeServices` generates a `RememberMeAuthenticationToken`, which is processed by `RememberMeAuthenticationProvider`. A `key` is shared between this authentication provider and the `TokenBasedRememberMeServices`. In addition, `TokenBasedRememberMeServices` requires A UserDetailsService from which it can retrieve the username and password for signature comparison purposes, and generate the `RememberMeAuthenticationToken` to contain the correct `GrantedAuthority`[]s. Some sort of

[1] Essentially, the username is not included in the cookie, to prevent exposing a valid login name unecessarily. There is a discussion on this in the comments section of this article.

logout command should be provided by the application that invalidates the cookie if the user requests this. `TokenBasedRememberMeServices` also implements Spring Security's `LogoutHandler` interface so can be used with `LogoutFilter` to have the cookie cleared automatically.

The beans required in an application context to enable remember-me services are as follows:

```
<bean id="rememberMeProcessingFilter"
    class="org.springframework.security.ui.rememberme.RememberMeProcessingFilter">
  <property name="rememberMeServices" ref="rememberMeServices"/>
  <property name="authenticationManager" ref="theAuthenticationManager" />
</bean>

<bean id="rememberMeServices" class="org.springframework.security.ui.rememberme.TokenBasedRememberMeServices">
  <property name="userDetailsService" ref="myUserDetailsService"/>
  <property name="key" value="springRocks"/>
</bean>

<bean id="rememberMeAuthenticationProvider"
    class="org.springframework.security.providers.rememberme.RememberMeAuthenticationProvider">
  <property name="key" value="springRocks"/>
</bean>
```

Don't forget to add your `RememberMeServices` implementation to your `AuthenticationProcessingFilter.setRememberMeServices()` property, include the `RememberMeAuthenticationProvider` in your `AuthenticationManager.setProviders()` list, and add `RememberMeProcessingFilter` into your `FilterChainProxy` (typically immediately after your `AuthenticationProcessingFilter`).

14.4.2. PersistentTokenBasedRememberMeServices

This class can be used in the same way as `TokenBasedRememberMeServices`, but it additionally needs to be configured with a `PersistentTokenRepository` to store the tokens. There are two standard implementations.

- `InMemoryTokenRepositoryImpl` which is intended for testing only.
- `JdbcTokenRepositoryImpl` which stores the tokens in a database.

The database schema is described above in Section 14.3, “Persistent Token Approach”.

Chapter 15. Java Authentication and Authorization Service (JAAS) Provider

15.1. Overview

Spring Security provides a package able to delegate authentication requests to the Java Authentication and Authorization Service (JAAS). This package is discussed in detail below.

Central to JAAS operation are login configuration files. To learn more about JAAS login configuration files, consult the JAAS reference documentation available from Sun Microsystems. We expect you to have a basic understanding of JAAS and its login configuration file syntax in order to understand this section.

15.2. Configuration

The `JaasAuthenticationProvider` attempts to authenticate a user's principal and credentials through JAAS.

Let's assume we have a JAAS login configuration file, `/WEB-INF/login.conf`, with the following contents:

```
JAASTest {
    sample.SampleLoginModule required;
};
```

Like all Spring Security beans, the `JaasAuthenticationProvider` is configured via the application context. The following definitions would correspond to the above JAAS login configuration file:

```
<bean id="jaasAuthenticationProvider"
           class="org.springframework.security.providers.jaas.JaasAuthenticationProvider">
  <property name="loginConfig" value="/WEB-INF/login.conf"/>
  <property name="loginContextName" value="JAASTest"/>
  <property name="callbackHandlers">
    <list>
      <bean class="org.springframework.security.providers.jaas.JaasNameCallbackHandler"/>
      <bean class="org.springframework.security.providers.jaas.JaasPasswordCallbackHandler"/>
    </list>
  </property>
  <property name="authorityGranters">
    <list>
      <bean class="org.springframework.security.providers.jaas.TestAuthorityGranter"/>
    </list>
  </property>
</bean>
```

The `CallbackHandlers` and `AuthorityGranters` are discussed below.

15.2.1. JAAS CallbackHandler

Most JAAS `LoginModules` require a callback of some sort. These callbacks are usually used to obtain the username and password from the user.

In a Spring Security deployment, Spring Security is responsible for this user interaction (via the authentication mechanism). Thus, by the time the authentication request is delegated through to JAAS, Spring Security's authentication mechanism will already have fully-populated an `Authentication` object containing all the

information required by the JAAS `LoginModule`.

Therefore, the JAAS package for Spring Security provides two default callback handlers, `JaasNameCallbackHandler` and `JaasPasswordCallbackHandler`. Each of these callback handlers implement `JaasAuthenticationCallbackHandler`. In most cases these callback handlers can simply be used without understanding the internal mechanics.

For those needing full control over the callback behavior, internally `JaasAutheticationProvider` wraps these `JaasAuthenticationCallbackHandlers` with an `InternalCallbackHandler`. The `InternalCallbackHandler` is the class that actually implements JAAS' normal `CallbackHandler` interface. Any time that the JAAS `LoginModule` is used, it is passed a list of application context configured `InternalCallbackHandlers`. If the `LoginModule` requests a callback against the `InternalCallbackHandlers`, the callback is in-turn passed to the `JaasAuthenticationCallbackHandlers` being wrapped.

15.2.2. JAAS AuthorityGranter

JAAS works with principals. Even "roles" are represented as principals in JAAS. Spring Security, on the other hand, works with `Authentication` objects. Each `Authentication` object contains a single principal, and multiple `GrantedAuthority`[]s. To facilitate mapping between these different concepts, Spring Security's JAAS package includes an `AuthorityGranter` interface.

An `AuthorityGranter` is responsible for inspecting a JAAS principal and returning a `String`. The `JaasAuthenticationProvider` then creates a `JaasGrantedAuthority` (which implements Spring Security's `GrantedAuthority` interface) containing both the `AuthorityGranter`-returned `String` and the JAAS principal that the `AuthorityGranter` was passed. The `JaasAuthenticationProvider` obtains the JAAS principals by firstly successfully authenticating the user's credentials using the JAAS `LoginModule`, and then accessing the `LoginContext` it returns. A call to `LoginContext.getSubject().getPrincipals()` is made, with each resulting principal passed to each `AuthorityGranter` defined against the `JaasAuthenticationProvider.setAuthorityGranters(List)` property.

Spring Security does not include any production `AuthorityGranters` given that every JAAS principal has an implementation-specific meaning. However, there is a `TestAuthorityGranter` in the unit tests that demonstrates a simple `AuthorityGranter` implementation.

Chapter 16. Pre-Authentication Scenarios

There are situations where you want to use Spring Security for authorization, but the user has already been reliably authenticated by some external system prior to accessing the application. We refer to these situations as “pre-authenticated” scenarios. Examples include X.509, Siteminder and authentication by the J2EE container in which the application is running. When using pre-authentication, Spring Security has to

1. Identify the user making the request.

2. Obtain the authorities for the user.

The details will depend on the external authentication mechanism. A user might be identified by their certificate information in the case of X.509, or by an HTTP request header in the case of Siteminder. If relying on container authentication, the user will be identified by calling the `getUserPrincipal()` method on the incoming HTTP request. In some cases, the external mechanism may supply role/authority information for the user but in others the authorities must be obtained from a separate source, such as a `UserDetailsService`.

16.1. Pre-Authentication Framework Classes

Because most pre-authentication mechanisms follow the same pattern, Spring Security has a set of classes which provide an internal framework for implementing pre-authenticated authentication providers. This removes duplication and allows new implementations to be added in a structured fashion, without having to write everything from scratch. You don't need to know about these classes if you want to use something like X.509 authentication, as it already has a namespace configuration option which is simpler to use and get started with. If you need to use explicit bean confiuration or are planning on writing your own implementation then an understanding of how the provided implementations work will be useful. You will find the web related classes under the org.springframework.security.ui.preauth package and the backend classes under org.springframework.security.providers.preauth. We just provide an outline here so you should consult the Javadoc and source where appropriate.

16.1.1. AbstractPreAuthenticatedProcessingFilter

This class will check the current contents of the security context and, if empty, it will attempt to extract user information from the HTTP request and submit it to the `AuthenticationManager`. Subclasses override the following methods to obtain this information:

```
protected abstract Object getPreAuthenticatedPrincipal(HttpServletRequest request);

protected abstract Object getPreAuthenticatedCredentials(HttpServletRequest request);
```

After calling these, the filter will create a `PreAuthenticatedAuthenticationToken` containing the returned data and submit it for authentication. By “authentication” here, we really just mean further processing to perhaps load the user's authorities, but the standard Spring Security authentication architecture is followed.

16.1.2. AbstractPreAuthenticatedAuthenticationDetailsSource

Like other Spring Security authentication filters, the pre-authentication filter has an `authenticationDetailsSource` property which by default will create a `WebAuthenticationDetails` object to store additional information such as the session-identifier and originating IP address in the `details` property of the `Authentication` object. In cases where user role information can be obtained from the pre-authentication

mechanism, the data is also stored in this property. Subclasses of `AbstractPreAuthenticatedAuthenticationDetailsSource` use an extended details object which implements the `GrantedAuthoritiesContainer` interface, thus enabling the authentication provider to read the authorities which were externally allocated to the user. We'll look at a concrete example next.

16.1.2.1. J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource

If the filter is configured with an `authenticationDetailsSource` which is an instance of this class, the authority information is obtained by calling the `isUserInRole(String role)` method for each of a pre-determined set of “mappable roles”. The class gets these from a configured `MappableAttributesRetriever`. Possible implementations include hard-coding a list in the application context and reading the role information from the `<security-role>` information in a `web.xml` file. The pre-authentication sample application uses the latter approach.

There is an additional stage where the roles (or attributes) are mapped to Spring Security `GrantedAuthority` objects using a configured `Attributes2GrantedAuthoritiesMapper`. The default will just add the usual `ROLE_` prefix to the names, but it gives you full control over the behaviour.

16.1.3. PreAuthenticatedAuthenticationProvider

The pre-authenticated provider has little more to do than load the `UserDetails` object for the user. It does this by delegating to a `AuthenticationUserDetailsService`. The latter is similar to the standard `UserDetailsService` but takes an `Authentication` object rather than just user name:

```
public interface AuthenticationUserDetailsService {
  UserDetails loadUserDetails(Authentication token) throws UsernameNotFoundException;
}
```

This interface may have also other uses but with pre-authentication it allows access to the authorities which were packaged in the `Authentication` object, as we saw in the previous section. The `PreAuthenticatedGrantedAuthoritiesUserDetailsService` class does this. Alternatively, it may delegate to a standard `UserDetailsService` via the `UserDetailsByNameServiceWrapper` implementation.

16.1.4. PreAuthenticatedProcessingFilterEntryPoint

The `AuthenticationEntryPoint` was discussed in the technical overview chapter. Normally it is responsible for kick-starting the authentication process for an unauthenticated user (when they try to access a protected resource), but in the pre-authenticated case this doesn't apply. You would only configure the `ExceptionTranslationFilter` with an instance of this class if you aren't using pre-authentication in combination with other authentication mechanisms. It will be called if the user is rejected by the `AbstractPreAuthenticatedProcessingFilter` resulting in a null authentication. It always returns a `403`-forbidden response code if called.

16.2. Concrete Implementations

X.509 authentication is covered in its own chapter. Here we'll look at some classes which provide support for other pre-authenticated scenarios.

16.2.1. Request-Header Authentication (Siteminder)

An external authentication system may supply information to the application by setting specific headers on the

HTTP request. A well known example of this is is Siteminder, which passes the username in a header called `SM_USER`. This mechanism is supported by the class `RequestHeaderPreAuthenticatedProcessingFilter` which simply extracts the username from the header. It defaults to using the name `SM_USER` as the header name. See the Javadoc for more details.

Tip

Note that when using a system like this, the framework performs no authentication checks at all and it is *extremely* important that the external system is configured properly and protects all access to the application. If an attacker is able to forge the headers in their original request without this being detected then they could potentially choose any userame they wished.

16.2.1.1. Siteminder Example Configuration

A typical configuration using this filter would look like this:

```
      <bean id="siteminderFilter"
    class="org.springframework.security.ui.preauth.header.RequestHeaderPreAuthenticatedProcessingFilter">
  <security:custom-filter position="PRE_AUTH_FILTER" />
  <property name="principalRequestHeader" value="SM_USER"/>
  <property name="authenticationManager" ref="authenticationManager" />
</bean>

<bean id="preauthAuthProvider"
    class="org.springframework.security.providers.preauth.PreAuthenticatedAuthenticationProvider">
  <security:custom-authentication-provider />
  <property name="preAuthenticatedUserDetailsService">
    <bean id="userDetailsServiceWrapper"
          class="org.springframework.security.userdetails.UserDetailsByNameServiceWrapper">
      <property name="userDetailsService" ref="userDetailsService"/>
    </bean>
  </property>
      </bean>

      <security:authentication-manager alias="authenticationManager" />
```

We've assumed here that the security namespace is being used for configuration (hence the user of the `custom-filter`, `authentication-manager` and `custom-authentication-provider` elements (you can read more about them in the namespace chapter). You would leave these out of a traditional bean configuration. It's also assumed that you have added a `UserDetailsService` (called “userDetailsService”) to your configuration to load the user's roles.

16.2.2. J2EE Container Authentication

The class `J2eePreAuthenticatedProcessingFilter` will extract the username from the `userPrincipal` property of the `HttpServletRequest`. use of this filter would usually be combined with the use of J2EE roles as described above in Section 16.1.2.1, “J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource”.

Chapter 17. Anonymous Authentication

17.1. Overview

Particularly in the case of web request URI security, sometimes it is more convenient to assign configuration attributes against every possible secure object invocation. Put differently, sometimes it is nice to say `ROLE_SOMETHING` is required by default and only allow certain exceptions to this rule, such as for login, logout and home pages of an application. There are also other situations where anonymous authentication would be desired, such as when an auditing interceptor queries the `SecurityContextHolder` to identify which principal was responsible for a given operation. Such classes can be authored with more robustness if they know the `SecurityContextHolder` always contains an `Authentication` object, and never `null`.

17.2. Configuration

Spring Security provides three classes that together provide an anonymous authentication feature. `AnonymousAuthenticationToken` is an implementation of `Authentication`, and stores the `GrantedAuthority`[]s which apply to the anonymous principal. There is a corresponding `AnonymousAuthenticationProvider`, which is chained into the `ProviderManager` so that `AnonymousAuthenticationTokens` are accepted. Finally, there is an AnonymousProcessingFilter, which is chained after the normal authentication mechanisms and automatically add an `AnonymousAuthenticationToken` to the `SecurityContextHolder` if there is no existing `Authentication` held there. The definition of the filter and authentication provider appears as follows:

```
<bean id="anonymousProcessingFilter"
    class="org.springframework.security.providers.anonymous.AnonymousProcessingFilter">
  <property name="key" value="foobar"/>
  <property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
</bean>

<bean id="anonymousAuthenticationProvider"
    class="org.springframework.security.providers.anonymous.AnonymousAuthenticationProvider">
  <property name="key" value="foobar"/>
</bean>
```

The `key` is shared between the filter and authentication provider, so that tokens created by the former are accepted by the latter. The `userAttribute` is expressed in the form of `usernameInTheAuthenticationToken,grantedAuthority[,grantedAuthority]`. This is the same syntax as used after the equals sign for `InMemoryDaoImpl`'s `userMap` property.

As explained earlier, the benefit of anonymous authentication is that all URI patterns can have security applied to them. For example:

```
<bean id="filterInvocationInterceptor"
    class="org.springframework.security.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/>
  <property name="objectDefinitionSource">
    <security:filter-invocation-definition-source>
      <security:intercept-url pattern='/index.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
      <security:intercept-url pattern='/hello.htm' access='ROLE_ANONYMOUS,ROLE_USER'/>
```

```
      <security:intercept-url pattern='/logoff.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
      <security:intercept-url pattern='/login.jsp' access='ROLE_ANONYMOUS,ROLE_USER'/>
      <security:intercept-url pattern='/**' access='ROLE_USER'/>
    </security:filter-invocation-definition-source>" +
  </property>
</bean>
```

Rounding out the anonymous authentication discussion is the `AuthenticationTrustResolver` interface, with its corresponding `AuthenticationTrustResolverImpl` implementation. This interface provides an `isAnonymous(Authentication)` method, which allows interested classes to take into account this special type of authentication status. The `ExceptionTranslationFilter` uses this interface in processing `AccessDeniedExceptions`. If an `AccessDeniedException` is thrown, and the authentication is of an anonymous type, instead of throwing a 403 (forbidden) response, the filter will instead commence the `AuthenticationEntryPoint` so the principal can authenticate properly. This is a necessary distinction, otherwise principals would always be deemed "authenticated" and never be given an opportunity to login via form, basic, digest or some other normal authentication mechanism

Chapter 18. X.509 Authentication

18.1. Overview

The most common use of X.509 certificate authentication is in verifying the identity of a server when using SSL, most commonly when using HTTPS from a browser. The browser will automatically check that the certificate presented by a server has been issued (ie digitally signed) by one of a list of trusted certificate authorities which it maintains.

You can also use SSL with “mutual authentication”; the server will then request a valid certificate from the client as part of the SSL handshake. The server will authenticate the client by checking that it's certificate is signed by an acceptable authority. If a valid certificate has been provided, it can be obtained through the servlet API in an application. Spring Security X.509 module extracts the certificate using a filter and passes it to the configured X.509 authentication provider to allow any additional application-specific checks to be applied. It also maps the certificate to an application user and loads that user's set of granted authorities for use with the standard Spring Security infrastructure.

You should be familiar with using certificates and setting up client authentication for your servlet container before attempting to use it with Spring Security. Most of the work is in creating and installing suitable certificates and keys. For example, if you're using Tomcat then read the instructions here `http://tomcat.apache.org/tomcat-6.0-doc/ssl-howto.html`. It's important that you get this working before trying it out with Spring Security

18.2. Adding X.509 Authentication to Your Web Application

Enabling X.509 client authentication is very straightforward. Just add the `<x509/>` element to your http security namespace configuration.

```
<http>
 ...
    <x509 subject-principal-regex="CN=(.*?)," user-service-ref="userService"/>
 ...
</http>
```

The element has two optional attributes:

- `subject-principal-regex`. The regular expression used to extract a username from the certificate's subject name. The default value is shown above. This is the username which will be passed to the `UserDetailsService` to load the authorities for the user.
- `user-service-ref`. This is the bean Id of the `UserDetailsService` to be used with X.509. It isn't needed if there is only one defined in your application context.

The `subject-principal-regex` should contain a single group. For example the default expression "CN=(.*?)," matches the common name field. So if the subject name in the certificate is "CN=Jimi Hendrix, OU=...", this will give a user name of "Jimi Hendrix". The matches are case insensitive. So "emailAddress=(.?)," will match "EMAILADDRESS=jimi@hendrix.org,CN=..." giving a user name "jimi@hendrix.org". If the client presents a certificate and a valid username is successfully extracted, then there should be a valid `Authentication` object in the security context. If no certificate is found, or no corresponding user could be found then the security context will remain empty. This means that you can easily use X.509 authentication with other options such as

a form-based login.

18.3. Setting up SSL in Tomcat

There are some pre-generated certificates in the `samples/certificate` directory in the Spring Security project. You can use these to enable SSL for testing if you don't want to generate your own. The file `server.jks` contains the server certificate, private key and the issuing certificate authority certificate. There are also some client certificate files for the users from the sample applications. You can install these in your browser to enable SSL client authentication.

To run tomcat with SSL support, drop the `server.jks` file into the tomcat `conf` directory and add the following connector to the `server.xml` file

```
<Connector port="8443" protocol="HTTP/1.1" SSLEnabled="true" scheme="https" secure="true"
           clientAuth="true" sslProtocol="TLS"
           keystoreFile="${catalina.home}/conf/server.jks"
           keystoreType="JKS" keystorePass="password"
           truststoreFile="${catalina.home}/conf/server.jks"
           truststoreType="JKS" truststorePass="password"
/>
```

`clientAuth` can also be set to *`want`* if you still want SSL connections to succeed even if the client doesn't provide a certificate. Clients which don't present a certificate won't be able to access any objects secured by Spring Security unless you use a non-X.509 authentication mechanism, such as form authentication.

Chapter 19. CAS Authentication

19.1. Overview

JA-SIG produces an enterprise-wide single sign on system known as CAS. Unlike other initiatives, JA-SIG's Central Authentication Service is open source, widely used, simple to understand, platform independent, and supports proxy capabilities. Spring Security fully supports CAS, and provides an easy migration path from single-application deployments of Spring Security through to multiple-application deployments secured by an enterprise-wide CAS server.

You can learn more about CAS at `http://www.ja-sig.org/products/cas/`. You will also need to visit this site to download the CAS Server files.

19.2. How CAS Works

Whilst the CAS web site contains documents that detail the architecture of CAS, we present the general overview again here within the context of Spring Security. Spring Security 2.0 supports CAS 3. At the time of writing, the CAS server was at version 3.2.

Somewhere in your enterprise you will need to setup a CAS server. The CAS server is simply a standard WAR file, so there isn't anything difficult about setting up your server. Inside the WAR file you will customise the login and other single sign on pages displayed to users.

When deploying a CAS 3.2 server, you will also need to specify an `AuthenticationHandler` in the `deployerConfigContext.xml` included with CAS. The `AuthenticationHandler` has a simple method that returns a boolean as to whether a given set of Credentials is valid. Your `AuthenticationHandler` implementation will need to link into some type of backend authentication repository, such as an LDAP server or database. CAS itself includes numerous `AuthenticationHandlers` out of the box to assist with this. When you download and deploy the server war file, it is set up to successfully authenticate users who enter a password matching their username, which is useful for testing.

Apart from the CAS server itself, the other key players are of course the secure web applications deployed throughout your enterprise. These web applications are known as "services". There are two types of services: standard services and proxy services. A proxy service is able to request resources from other services on behalf of the user. This will be explained more fully later.

19.3. Configuration of CAS Client

The web application side of CAS is made easy due to Spring Security. It is assumed you already know the basics of using Spring Security, so these are not covered again below. We'll assume a namespace based configuration is being used and add in the CAS beans as required.

You will need to add a `ServiceProperties` bean to your application context. This represents your service:

```
<bean id="serviceProperties" class="org.springframework.security.ui.cas.ServiceProperties">
  <property name="service" value="https://localhost:8443/cas-sample/j_spring_cas_security_check"/>
  <property name="sendRenew" value="false"/>
</bean>
```

The `service` must equal a URL that will be monitored by the `CasProcessingFilter`. The `sendRenew` defaults to false, but should be set to true if your application is particularly sensitive. What this parameter does is tell the CAS login service that a single sign on login is unacceptable. Instead, the user will need to re-enter their username and password in order to gain access to the service.

The following beans should be configured to commence the CAS authentication process:

```
<security:authentication-manager alias="authenticationManager"/>

<bean id="casProcessingFilter" class="org.springframework.security.ui.cas.CasProcessingFilter">
  <security:custom-filter after="CAS_PROCESSING_FILTER"/>
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="authenticationFailureUrl" value="/casfailed.jsp"/>
  <property name="defaultTargetUrl" value="/"/>
</bean>

<bean id="casProcessingFilterEntryPoint"
    class="org.springframework.security.ui.cas.CasProcessingFilterEntryPoint">
  <property name="loginUrl" value="https://localhost:9443/cas/login"/>
  <property name="serviceProperties" ref="serviceProperties"/>
</bean>
```

The `CasProcessingFilterEntryPoint` should be selected to drive authentication using `entry-point-ref`.

The `CasProcessingFilter` has very similar properties to the `AuthenticationProcessingFilter` (used for form-based logins). Each property is self-explanatory. Note that we've also used the namespace syntax for setting up an alias to the authentication mnager, since the `CasProcessingFilter` needs a reference to it.

For CAS to operate, the `ExceptionTranslationFilter` must have its `authenticationEntryPoint` property set to the `CasProcessingFilterEntryPoint` bean.

The `CasProcessingFilterEntryPoint` must refer to the `ServiceProperties` bean (discussed above), which provides the URL to the enterprise's CAS login server. This is where the user's browser will be redirected.

Next you need to add a `CasAuthenticationProvider` and its collaborators:

```
<bean id="casAuthenticationProvider" class="org.springframework.security.providers.cas.CasAuthenticationProvid
  <security:custom-authentication-provider />
  <property name="userDetailsService" ref="userService"/>
  <property name="serviceProperties" ref="serviceProperties" />
  <property name="ticketValidator">
    <bean class="org.jasig.cas.client.validation.Cas20ServiceTicketValidator">
      <constructor-arg index="0" value="https://localhost:9443/cas" />
      </bean>
  </property>
  <property name="key" value="an_id_for_this_auth_provider_only"/>
</bean>

<security:user-service id="userService">
  <security:user name="joe" password="joe" authorities="ROLE_USER" />
  ...
</security:user-service>
```

The `CasAuthenticationProvider` uses a `UserDetailsService` instance to load the authorities for a user, once they have been authentiated by CAS. We've shown a simple in-memory setup here.

The beans are all reasonable self-explanatory if you refer back to the "How CAS Works" section.

Chapter 20. Run-As Authentication Replacement

20.1. Overview

The AbstractSecurityInterceptor is able to temporarily replace the Authentication object in the SecurityContext and SecurityContextHolder during the secure object callback phase. This only occurs if the original Authentication object was successfully processed by the AuthenticationManager and AccessDecisionManager. The RunAsManager will indicate the replacement Authentication object, if any, that should be used during the SecurityInterceptorCallback.

By temporarily replacing the Authentication object during the secure object callback phase, the secured invocation will be able to call other objects which require different authentication and authorization credentials. It will also be able to perform any internal security checks for specific GrantedAuthority objects. Because Spring Security provides a number of helper classes that automatically configure remoting protocols based on the contents of the SecurityContextHolder, these run-as replacements are particularly useful when calling remote web services

20.2. Configuration

A RunAsManager interface is provided by Spring Security:

```
Authentication buildRunAs(Authentication authentication, Object object, ConfigAttributeDefinition config);
boolean supports(ConfigAttribute attribute);
boolean supports(Class clazz);
```

The first method returns the Authentication object that should replace the existing Authentication object for the duration of the method invocation. If the method returns null, it indicates no replacement should be made. The second method is used by the AbstractSecurityInterceptor as part of its startup validation of configuration attributes. The supports(Class) method is called by a security interceptor implementation to ensure the configured RunAsManager supports the type of secure object that the security interceptor will present.

One concrete implementation of a RunAsManager is provided with Spring Security. The RunAsManagerImpl class returns a replacement RunAsUserToken if any ConfigAttribute starts with RUN_AS_. If any such ConfigAttribute is found, the replacement RunAsUserToken will contain the same principal, credentials and granted authorities as the original Authentication object, along with a new GrantedAuthorityImpl for each RUN_AS_ ConfigAttribute. Each new GrantedAuthorityImpl will be prefixed with ROLE_, followed by the RUN_AS ConfigAttribute. For example, a RUN_AS_SERVER will result in the replacement RunAsUserToken containing a ROLE_RUN_AS_SERVER granted authority.

The replacement RunAsUserToken is just like any other Authentication object. It needs to be authenticated by the AuthenticationManager, probably via delegation to a suitable AuthenticationProvider. The RunAsImplAuthenticationProvider performs such authentication. It simply accepts as valid any RunAsUserToken presented.

To ensure malicious code does not create a RunAsUserToken and present it for guaranteed acceptance by the RunAsImplAuthenticationProvider, the hash of a key is stored in all generated tokens. The RunAsManagerImpl and RunAsImplAuthenticationProvider is created in the bean context with the same key:

```

```

```
<bean id="runAsManager" class="org.springframework.security.runas.RunAsManagerImpl">
  <property name="key" value="my_run_as_password"/>
</bean>

<bean id="runAsAuthenticationProvider"
    class="org.springframework.security.runas.RunAsImplAuthenticationProvider">
  <property name="key" value="my_run_as_password"/>
</bean>
```

By using the same key, each `RunAsUserToken` can be validated it was created by an approved `RunAsManagerImpl`. The `RunAsUserToken` is immutable after creation for security reasons

Chapter 21. Container Adapter Authentication

21.1. Overview

Very early versions of Spring Security exclusively used Container Adapters for interfacing authentication with end users. Whilst this worked well, it required considerable time to support multiple container versions and the configuration itself was relatively time-consuming for developers. For this reason the HTTP Form Authentication and HTTP Basic Authentication approaches were developed, and are today recommended for almost all applications.

Container Adapters enable Spring Security to integrate directly with the containers used to host end user applications. This integration means that applications can continue to leverage the authentication and authorization capabilities built into containers (such as `isUserInRole()` and form-based or basic authentication), whilst benefiting from the enhanced security interception capabilities provided by Spring Security (it should be noted that Spring Security also offers `ContextHolderAwareRequestWrapper` to deliver `isUserInRole()` and similar Servlet Specification compatibility methods).

The integration between a container and Spring Security is achieved through an adapter. The adapter provides a container-compatible user authentication provider, and needs to return a container-compatible user object.

The adapter is instantiated by the container and is defined in a container-specific configuration file. The adapter then loads a Spring application context which defines the normal authentication manager settings, such as the authentication providers that can be used to authenticate the request. The application context is usually named `acegisecurity.xml` and is placed in a container-specific location.

Spring Security currently supports Jetty, Catalina (Tomcat), JBoss and Resin. Additional container adapters can easily be written

21.2. Adapter Authentication Provider

As is always the case, the container adapter generated `Authentication` object still needs to be authenticated by an `AuthenticationManager` when requested to do so by the `AbstractSecurityInterceptor`. The `AuthenticationManager` needs to be certain the adapter-provided `Authentication` object is valid and was actually authenticated by a trusted adapter.

Adapters create `Authentication` objects which are immutable and implement the `AuthByAdapter` interface. These objects store the hash of a key that is defined by the adapter. This allows the `Authentication` object to be validated by the `AuthByAdapterProvider`. This authentication provider is defined as follows:

```
<bean id="authByAdapterProvider"
        class="org.springframework.security.adapters.AuthByAdapterProvider">
<property name="key"><value>my_password</value></property>
</bean>
```

The key must match the key that is defined in the container-specific configuration file that starts the adapter. The `AuthByAdapterProvider` automatically accepts as valid any `AuthByAdapter` implementation that returns the expected hash of the key.

To reiterate, this means the adapter will perform the initial authentication using providers such as `DaoAuthenticationProvider`, returning an `AuthByAdapter` instance that contains a hash code of the key. Later,

when an application calls a security interceptor managed resource, the `AuthByAdapter` instance in the `SecurityContext` in the `SecurityContextHolder` will be tested by the application's `AuthByAdapterProvider`. There is no requirement for additional authentication providers such as `DaoAuthenticationProvider` within the application-specific application context, as the only type of `Authentication` instance that will be presented by the application is from the container adapter.

Classloader issues are frequent with containers and the use of container adapters illustrates this further. Each container requires a very specific configuration. The installation instructions are provided below. Once installed, please take the time to try the sample application to ensure your container adapter is properly configured.

When using container adapters with the `DaoAuthenticationProvider`, ensure you set its `forcePrincipalAsString` property to `true`.

21.3. Jetty

The following was tested with Jetty 4.2.18.

`$JETTY_HOME` refers to the root of your Jetty installation.

Edit your `$JETTY_HOME/etc/jetty.xml` file so the `<Configure class>` section has a new `addRealm` call:

```
<Call name="addRealm">
<Arg>
  <New class="org.springframework.security.adapters.jetty.JettySpringSecurityUserRealm">
    <Arg>Spring Powered Realm</Arg>
    <Arg>my_password</Arg>
    <Arg>etc/acegisecurity.xml</Arg>
  </New>
</Arg>
</Call>
```

Copy `acegisecurity.xml` into `$JETTY_HOME/etc`.

Copy the following files into `$JETTY_HOME/ext`:

- `aopalliance.jar`
- `commons-logging.jar`
- `spring.jar`
- `acegi-security-jetty-XX.jar`
- `commons-codec.jar`
- `burlap.jar`
- `hessian.jar`

None of the above JAR files (or `acegi-security-XX.jar`) should be in your application's `WEB-INF/lib`. The realm name indicated in your `web.xml` does matter with Jetty. The `web.xml` must express the same `<realm-name>` as your `jetty.xml` (in the example above, "Spring Powered Realm").

21.4. JBoss

The following was tested with JBoss 3.2.6.

`$JBOSS_HOME` refers to the root of your JBoss installation.

There are two different ways of making spring context available to the Jboss integration classes.

The first approach is by editing your `$JBOSS_HOME/server/your_config/conf/login-config.xml` file so that it contains a new entry under the `<Policy>` section:

```
<application-policy name = "SpringPoweredRealm">
<authentication>
  <login-module code = "org.springframework.security.adapters.jboss.JbossSpringSecurityLoginModule"
        flag = "required">
    <module-option name = "appContextLocation">acegisecurity.xml</module-option>
    <module-option name = "key">my_password</module-option>
 </login-module>
</authentication>
</application-policy>
```

Copy `acegisecurity.xml` into `$JBOSS_HOME/server/your_config/conf`.

In this configuration `acegisecurity.xml` contains the spring context definition including all the authentication manager beans. You have to bear in mind though, that `SecurityContext` is created and destroyed on each login request, so the login operation might become costly. Alternatively, the second approach is to use Spring singleton capabilities through `org.springframework.beans.factory.access.SingletonBeanFactoryLocator`. The required configuration for this approach is:

```
<application-policy name = "SpringPoweredRealm">
<authentication>
  <login-module code = "org.springframework.security.adapters.jboss.JbossSpringSecurityLoginModule"
        flag = "required">
    <module-option name = "singletonId">springRealm</module-option>
    <module-option name = "key">my_password</module-option>
    <module-option name = "authenticationManager">authenticationManager</module-option>
 </login-module>
</authentication>
</application-policy>
```

In the above code fragment, `authenticationManager` is a helper property that defines the expected name of the `AuthenticationManager` in case you have several defined in the IoC container. The `singletonId` property references a bean defined in a `beanRefFactory.xml` file. This file needs to be available from anywhere on the JBoss classpath, including `$JBOSS_HOME/server/your_config/conf`. The `beanRefFactory.xml` contains the following declaration:

```
<beans>
<bean id="springRealm" singleton="true" lazy-init="true" class="org.springframework.context.support.ClassPathXml
<constructor-arg>
  <list>
    <value>acegisecurity.xml</value>
```

```
  </list>
</constructor-arg>
</bean>
</beans>
```

Finally, irrespective of the configuration approach you need to copy the following files into `$JBOSS_HOME/server/your_config/lib`:

- `aopalliance.jar`
- `spring.jar`
- `acegi-security-jboss-XX.jar`
- `commons-codec.jar`
- `burlap.jar`
- `hessian.jar`

None of the above JAR files (or `acegi-security-XX.jar`) should be in your application's `WEB-INF/lib`. The realm name indicated in your `web.xml` does not matter with JBoss. However, your web application's `WEB-INF/jboss-web.xml` must express the same `<security-domain>` as your `login-config.xml`. For example, to match the above example, your `jboss-web.xml` would look like this:

```
<jboss-web>
<security-domain>java:/jaas/SpringPoweredRealm</security-domain>
</jboss-web>
```

JBoss is a widely-used container adapter (mostly due to the need to support legacy EJBs), so please let us know if you have any difficulties.

21.5. Resin

The following was tested with Resin 3.0.6.

`$RESIN_HOME` refers to the root of your Resin installation.

Resin provides several ways to support the container adapter. In the instructions below we have elected to maximise consistency with other container adapter configurations. This will allow Resin users to simply deploy the sample application and confirm correct configuration. Developers comfortable with Resin are naturally able to use its capabilities to package the JARs with the web application itself, and/or support single sign-on.

Copy the following files into `$RESIN_HOME/lib`:

- `aopalliance.jar`
- `commons-logging.jar`
- `spring.jar`

- acegi-security-resin-XX.jar
- commons-codec.jar
- burlap.jar
- hessian.jar

Unlike the container-wide acegisecurity.xml files used by other container adapters, each Resin web application will contain its own WEB-INF/resin-acegisecurity.xml file. Each web application will also contain a resin-web.xml file which Resin uses to start the container adapter:

```
<web-app>
<authenticator>
<type>org.springframework.security.adapters.resin.ResinAcegiAuthenticator</type>
<init>
  <app-context-location>WEB-INF/resin-acegisecurity.xml</app-context-location>
  <key>my_password</key>
</init>
</authenticator>
</web-app>
```

With the basic configuration provided above, none of the JAR files listed (or acegi-security-XX.jar) should be in your application's WEB-INF/lib. The realm name indicated in your web.xml does not matter with Resin, as the relevant authentication class is indicated by the <authenticator> setting

21.6. Tomcat

The following was tested with Jakarta Tomcat 4.1.30 and 5.0.19.

$CATALINA_HOME refers to the root of your Catalina (Tomcat) installation.

Edit your $CATALINA_HOME/conf/server.xml file so the <Engine> section contains only one active <Realm> entry. An example realm entry:

```
      <Realm
        className="org.springframework.security.adapters.catalina.CatalinaSpringSecurityUserRealm"
        appContextLocation="conf/acegisecurity.xml"
         key="my_password" />
```

Be sure to remove any other <Realm> entry from your <Engine> section.

Copy acegisecurity.xml into $CATALINA_HOME/conf.

Copy spring-security-catalina-XX.jar into $CATALINA_HOME/server/lib.

Copy the following files into $CATALINA_HOME/common/lib:

- aopalliance.jar
- spring.jar

- `commons-codec.jar`
- `burlap.jar`
- `hessian.jar`

None of the above JAR files (or `spring-security-XX.jar`) should be in your application's `WEB-INF/lib`. The realm name indicated in your `web.xml` does not matter with Catalina.

We have received reports of problems using this Container Adapter with Mac OS X. A work-around is to use a script such as follows:

```
#!/bin/sh
export CATALINA_HOME="/Library/Tomcat"
export JAVA_HOME="/Library/Java/Home"
cd /
$CATALINA_HOME/bin/startup.sh
```

Finally, restart Tomcat.

Part IV. Authorization

The advanced authorization capabilities within Spring Security represent one of the most compelling reasons for its popularity. Irrespective of how you choose to authenticate - whether using a Spring Security-provided mechanism and provider, or integrating with a container or other non-Spring Security authentication authority - you will find the authorization services can be used within your application in a consistent and simple way.

In this part we'll explore the different `AbstractSecurityInterceptor` implementations, which were introduced in Part I. We then move on to explore how to fine-tune authorization through use of domain access control lists.

Chapter 22. Common Authorization Concepts

22.1. Authorities

As briefly mentioned in the Authentication section, all `Authentication` implementations are required to store an array of `GrantedAuthority` objects. These represent the authorities that have been granted to the principal. The `GrantedAuthority` objects are inserted into the `Authentication` object by the `AuthenticationManager` and are later read by `AccessDecisionManagers` when making authorization decisions.

`GrantedAuthority` is an interface with only one method:

```
String getAuthority();
```

This method allows `AccessDecisionManagers` to obtain a precise `String` representation of the `GrantedAuthority`. By returning a representation as a `String`, a `GrantedAuthority` can be easily "read" by most `AccessDecisionManagers`. If a `GrantedAuthority` cannot be precisely represented as a `String`, the `GrantedAuthority` is considered "complex" and `getAuthority()` must return `null`.

An example of a "complex" `GrantedAuthority` would be an implementation that stores a list of operations and authority thresholds that apply to different customer account numbers. Representing this complex `GrantedAuthority` as a `String` would be quite complex, and as a result the `getAuthority()` method should return `null`. This will indicate to any `AccessDecisionManager` that it will need to specifically support the `GrantedAuthority` implementation in order to understand its contents.

Spring Security includes one concrete `GrantedAuthority` implementation, `GrantedAuthorityImpl`. This allows any user-specified `String` to be converted into a `GrantedAuthority`. All `AuthenticationProviders` included with the security architecture use `GrantedAuthorityImpl` to populate the `Authentication` object.

22.2. Pre-Invocation Handling

As we'll see in the Technical Overview chapter, Spring Security provides interceptors which control access to secure objects such as method invocations or web requests. A pre-invocation decision on whether the invocation is allowed to proceed is made by the `AccessDecisionManager`.

22.2.1. The AccessDecisionManager

The `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` and is responsible for making final access control decisions. The `AccessDecisionManager` interface contains three methods:

```
void decide(Authentication authentication, Object secureObject, ConfigAttributeDefinition config) throws Access
boolean supports(ConfigAttribute attribute);
boolean supports(Class clazz);
```

As can be seen from the first method, the `AccessDecisionManager` is passed via method parameters all information that is likely to be of value in assessing an authorization decision. In particular, passing the secure `Object` enables those arguments contained in the actual secure object invocation to be inspected. For example, let's assume the secure object was a `MethodInvocation`. It would be easy to query the `MethodInvocation` for any `Customer` argument, and then implement some sort of security logic in the `AccessDecisionManager` to

ensure the principal is permitted to operate on that customer. Implementations are expected to throw an `AccessDeniedException` if access is denied.

The `supports(ConfigAttribute)` method is called by the `AbstractSecurityInterceptor` at startup time to determine if the `AccessDecisionManager` can process the passed `ConfigAttribute`. The `supports(Class)` method is called by a security interceptor implementation to ensure the configured `AccessDecisionManager` supports the type of secure object that the security interceptor will present.

22.2.1.1. Voting-Based AccessDecisionManager Implementations

Whilst users can implement their own `AccessDecisionManager` to control all aspects of authorization, Spring Security includes several `AccessDecisionManager` implementations that are based on voting. Figure 22.1, "Voting Decision Manager" illustrates the relevant classes.

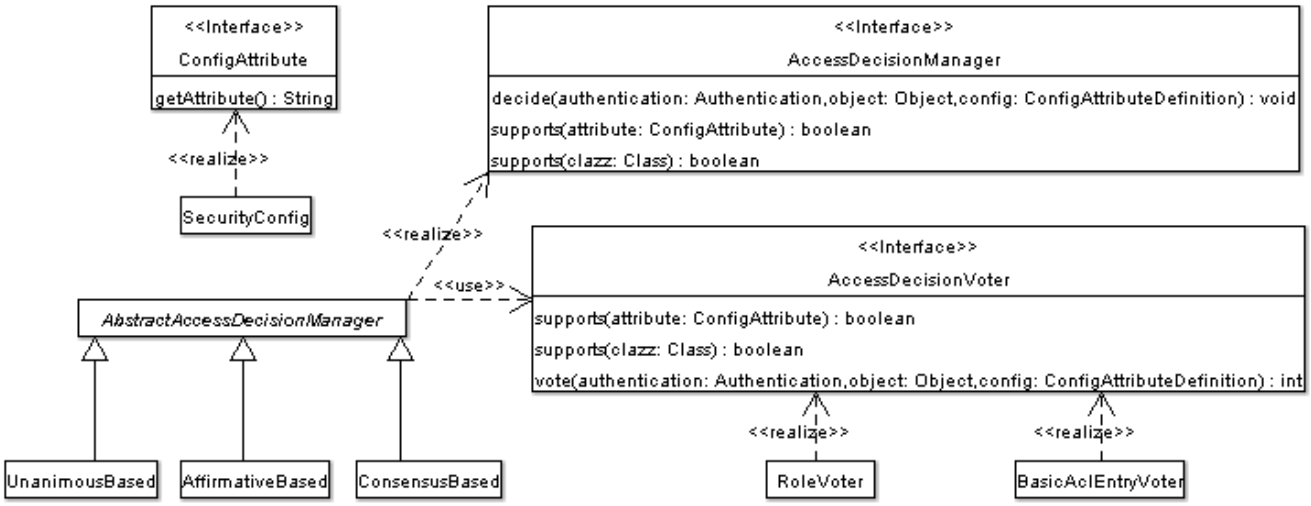

Figure 22.1. Voting Decision Manager

Using this approach, a series of `AccessDecisionVoter` implementations are polled on an authorization decision. The `AccessDecisionManager` then decides whether or not to throw an `AccessDeniedException` based on its assessment of the votes.

The `AccessDecisionVoter` interface has three methods:

```
int vote(Authentication authentication, Object object, ConfigAttributeDefinition config);
boolean supports(ConfigAttribute attribute);
boolean supports(Class clazz);
```

Concrete implementations return an `int`, with possible values being reflected in the `AccessDecisionVoter` static fields `ACCESS_ABSTAIN`, `ACCESS_DENIED` and `ACCESS_GRANTED`. A voting implementation will return `ACCESS_ABSTAIN` if it has no opinion on an authorization decision. If it does have an opinion, it must return either `ACCESS_DENIED` or `ACCESS_GRANTED`.

There are three concrete `AccessDecisionManagers` provided with Spring Security that tally the votes. The `ConsensusBased` implementation will grant or deny access based on the consensus of non-abstain votes. Properties are provided to control behavior in the event of an equality of votes or if all votes are abstain. The `AffirmativeBased` implementation will grant access if one or more `ACCESS_GRANTED` votes were received (i.e. a deny vote will be ignored, provided there was at least one grant vote). Like the `ConsensusBased` implementation, there is a parameter that controls the behavior if all voters abstain. The `UnanimousBased` provider expects unanimous `ACCESS_GRANTED` votes in order to grant access, ignoring abstains. It will deny access if there is any `ACCESS_DENIED` vote. Like the other implementations, there is a parameter that controls the

behaviour if all voters abstain.

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 may have a veto effect.

22.2.1.1.1. `RoleVoter`

The most commonly used `AccessDecisionVoter` provided with Spring Security is the simple `RoleVoter`, which treats configuration attributes as simple role names and votes to grant access if the user has been assigned that role.

It will vote if any ConfigAttribute begins with the prefix `ROLE_`. It will vote to grant access if there is a `GrantedAuthority` which returns a `String` representation (via the `getAuthority()` method) exactly equal to one or more `ConfigAttributes` starting with `ROLE_`. If there is no exact match of any `ConfigAttribute` starting with `ROLE_`, the `RoleVoter` will vote to deny access. If no `ConfigAttribute` begins with `ROLE_`, the voter will abstain. `RoleVoter` is case sensitive on comparisons as well as the `ROLE_` prefix.

22.2.1.1.2. Custom Voters

It is also possible to implement a custom `AccessDecisionVoter`. Several examples are provided in Spring Security unit tests, including `ContactSecurityVoter` and `DenyVoter`. The `ContactSecurityVoter` abstains from voting decisions where a `CONTACT_OWNED_BY_CURRENT_USER` `ConfigAttribute` is not found. If voting, it queries the `MethodInvocation` to extract the owner of the `Contact` object that is subject of the method call. It votes to grant access if the `Contact` owner matches the principal presented in the `Authentication` object. It could have just as easily compared the `Contact` owner with some `GrantedAuthority` the `Authentication` object presented. All of this is achieved with relatively few lines of code and demonstrates the flexibility of the authorization model.

22.3. After Invocation Handling

Whilst the `AccessDecisionManager` is called by the `AbstractSecurityInterceptor` before proceeding with the secure object invocation, some applications need a way of modifying the object actually returned by the secure object invocation. Whilst you could easily implement your own AOP concern to achieve this, Spring Security provides a convenient hook that has several concrete implementations that integrate with its ACL capabilities.

Figure 22.2, “After Invocation Implementation” illustrates Spring Security's `AfterInvocationManager` and its concrete implementations.

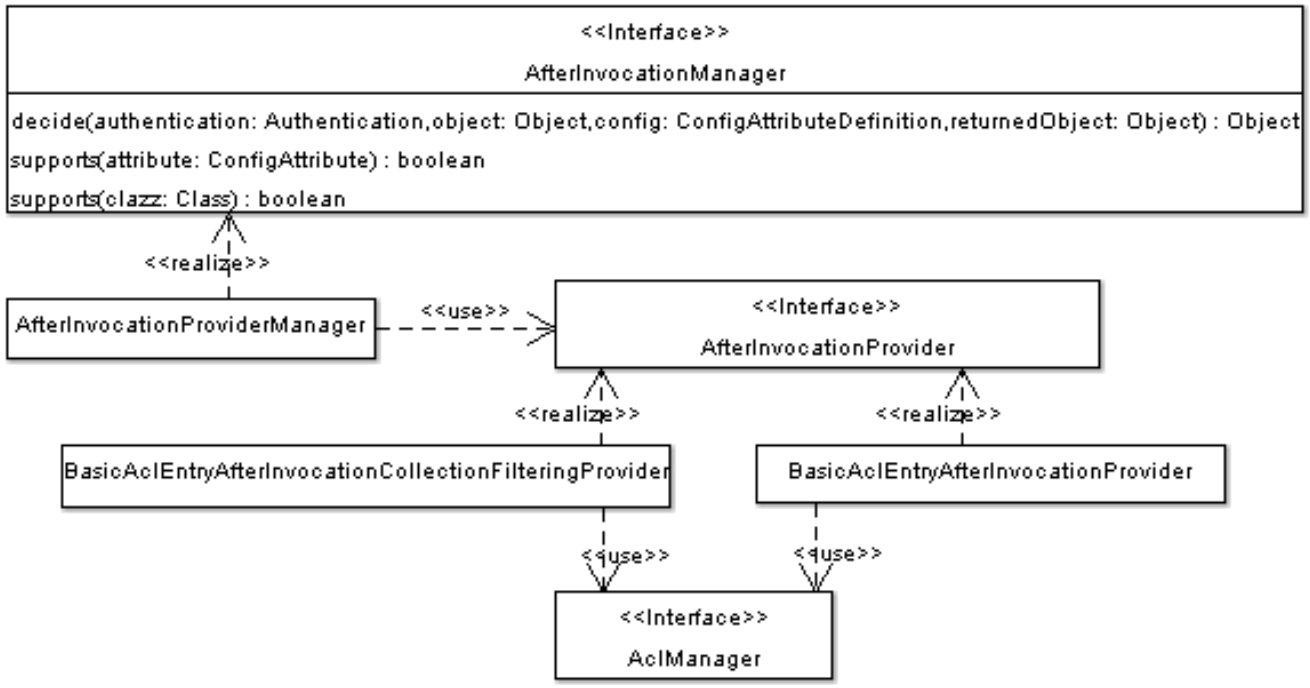

Figure 22.2. After Invocation Implementation

Like many other parts of Spring Security, `AfterInvocationManager` has a single concrete implementation, `AfterInvocationProviderManager`, which polls a list of `AfterInvocationProviders`. Each `AfterInvocationProvider` is allowed to modify the return object or throw an `AccessDeniedException`. Indeed multiple providers can modify the object, as the result of the previous provider is passed to the next in the list. Let's now consider our ACL-aware implementations of `AfterInvocationProvider`.

Please be aware that if you're using `AfterInvocationManager`, you will still need configuration attributes that allow the `MethodSecurityInterceptor`'s `AccessDecisionManager` to allow an operation. If you're using the typical Spring Security included `AccessDecisionManager` implementations, having no configuration attributes defined for a particular secure method invocation will cause each `AccessDecisionVoter` to abstain from voting. In turn, if the `AccessDecisionManager` property "`allowIfAllAbstainDecisions`" is `false`, an `AccessDeniedException` will be thrown. You may avoid this potential issue by either (i) setting "`allowIfAllAbstainDecisions`" to `true` (although this is generally not recommended) or (ii) simply ensure that there is at least one configuration attribute that an `AccessDecisionVoter` will vote to grant access for. This latter (recommended) approach is usually achieved through a `ROLE_USER` or `ROLE_AUTHENTICATED` configuration attribute

22.3.1. ACL-Aware AfterInvocationProviders

PLEASE NOTE: Acegi Security 1.0.3 contains a preview of a new ACL module. The new ACL module is a significant rewrite of the existing ACL module. The new module can be found under the `org.springframework.security.acls` package, with the old ACL module under `org.springframework.security.acl`. We encourage users to consider testing with the new ACL module and build applications with it. The old ACL module should be considered deprecated and may be removed from a future release. The following information relates to the new ACL package, and is thus recommended.

A common services layer method we've all written at one stage or another looks like this:

```
public Contact getById(Integer id);
```

Quite often, only principals with permission to read the `Contact` should be allowed to obtain it. In this situation the `AccessDecisionManager` approach provided by the `AbstractSecurityInterceptor` will not suffice. This is because the identity of the `Contact` is all that is available before the secure object is invoked. The `AclAfterInvocationProvider` delivers a solution, and is configured as follows:

```
<bean id="afterAclRead"
   class="org.springframework.security.afterinvocation.AclEntryAfterInvocationProvider">
  <constructor-arg ref="aclService"/>
  <constructor-arg>
    <list>
      <ref local="org.springframework.security.acls.domain.BasePermission.ADMINISTRATION"/>
      <ref local="org.springframework.security.acls.domain.BasePermission.READ"/>
    </list>
  </constructor-arg>
</bean>
```

In the above example, the `Contact` will be retrieved and passed to the `AclEntryAfterInvocationProvider`. The provider will thrown an `AccessDeniedException` if one of the listed `requirePermissions` is not held by the `Authentication`. The `AclEntryAfterInvocationProvider` queries the `AclService` to determine the ACL that applies for this domain object to this `Authentication`.

Similar to the `AclEntryAfterInvocationProvider` is `AclEntryAfterInvocationCollectionFilteringProvider`. It is designed to remove `Collection` or array elements for which a principal does not have access. It never thrown an `AccessDeniedException` - simply silently removes the offending elements. The provider is configured as follows:

```
<bean id="afterAclCollectionRead"
    class="org.springframework.security.afterinvocation.AclEntryAfterInvocationCollectionFilteringProvider">
  <constructor-arg ref="aclService"/>
  <constructor-arg>
    <list>
      <ref local="org.springframework.security.acls.domain.BasePermission.ADMINISTRATION"/>
      <ref local="org.springframework.security.acls.domain.BasePermission.READ"/>
    </list>
  </constructor-arg>
</bean>
```

As you can imagine, the returned `Object` must be a `Collection` or array for this provider to operate. It will remove any element if the `AclManager` indicates the `Authentication` does not hold one of the listed `requirePermissions`.

The Contacts sample application demonstrates these two `AfterInvocationProviders`.

22.3.2. ACL-Aware AfterInvocationProviders (old ACL module)

PLEASE NOTE: Acegi Security 1.0.3 contains a preview of a new ACL module. The new ACL module is a significant rewrite of the existing ACL module. The new module can be found under the `org.springframework.security.acls` package, with the old ACL module under `org.springframework.security.acl`. We encourage users to consider testing with the new ACL module and build applications with it. The old ACL module should be considered deprecated and may be removed from a future release.

A common services layer method we've all written at one stage or another looks like this:

```
public Contact getById(Integer id);
```

Quite often, only principals with permission to read the `Contact` should be allowed to obtain it. In this situation the `AccessDecisionManager` approach provided by the `AbstractSecurityInterceptor` will not suffice. This is because the identity of the `Contact` is all that is available before the secure object is invoked. The `BasicAclAfterInvocationProvider` delivers a solution, and is configured as follows:

```
<bean id="afterAclRead"
    class="org.springframework.security.afterinvocation.BasicAclEntryAfterInvocationProvider">
  <property name="aclManager" ref="aclManager"/>
  <property name="requirePermission">
    <list>
      <ref local="org.springframework.security.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="org.springframework.security.acl.basic.SimpleAclEntry.READ"/>
    </list>
  </property>
</bean>
```

In the above example, the `Contact` will be retrieved and passed to the `BasicAclEntryAfterInvocationProvider`. The provider will thrown an `AccessDeniedException` if one of the listed `requirePermissions` is not held by the `Authentication`. The `BasicAclEntryAfterInvocationProvider` queries the `AclManager` to determine the ACL that applies for this domain object to this `Authentication`.

Similar to the `BasicAclEntryAfterInvocationProvider` is `BasicAclEntryAfterInvocationCollectionFilteringProvider`. It is designed to remove `Collection` or array elements for which a principal does not have access. It never thrown an `AccessDeniedException` - simply silently removes the offending elements. The provider is configured as follows:

```
<bean id="afterAclCollectionRead"
    class="org.springframework.security.afterinvocation.BasicAclEntryAfterInvocationCollectionFilteringProvider"
  <property name="aclManager" ref="aclManager"/>
  <property name="requirePermission">
    <list>
      <ref local="org.springframework.security.acl.basic.SimpleAclEntry.ADMINISTRATION"/>
      <ref local="org.springframework.security.acl.basic.SimpleAclEntry.READ"/>
    </list>
  </property>
</bean>
```

As you can imagine, the returned `Object` must be a `Collection` or array for this provider to operate. It will remove any element if the `AclManager` indicates the `Authentication` does not hold one of the listed `requirePermissions`.

The Contacts sample application demonstrates these two `AfterInvocationProviders`.

22.4. Authorization Tag Libraries

`AuthorizeTag` is used to include content if the current principal holds certain `GrantedAuthoritys`.

The following JSP fragment illustrates how to use the `AuthorizeTag`:

```
<security:authorize ifAllGranted="ROLE_SUPERVISOR">
<td>
  <a href="del.htm?id=<c:out value="${contact.id}"/>">Del</a>
</td>
</security:authorize>
```

This tag would cause the tag's body to be output if the principal has been granted ROLE_SUPERVISOR.

The `security:authorize` tag declares the following attributes:

- `ifAllGranted`: All the listed roles must be granted for the tag to output its body.
- `ifAnyGranted`: Any of the listed roles must be granted for the tag to output its body.
- `ifNotGranted`: None of the listed roles must be granted for the tag to output its body.

You'll note that in each attribute you can list multiple roles. Simply separate the roles using a comma. The `authorize` tag ignores whitespace in attributes.

The tag library logically ANDs all of it's parameters together. This means that if you combine two or more attributes, all attributes must be true for the tag to output it's body. Don't add an `ifAllGranted="ROLE_SUPERVISOR"`, followed by an `ifNotGranted="ROLE_SUPERVISOR"`, or you'll be surprised to never see the tag's body.

By requiring all attributes to return true, the authorize tag allows you to create more complex authorization scenarios. For example, you could declare an `ifAllGranted="ROLE_SUPERVISOR"` and an `ifNotGranted="ROLE_NEWBIE_SUPERVISOR"` in the same tag, in order to prevent new supervisors from seeing the tag body. However it would no doubt be simpler to use `ifAllGranted="ROLE_EXPERIENCED_SUPERVISOR"` rather than inserting NOT conditions into your design.

One last item: the tag verifies the authorizations in a specific order: first `ifNotGranted`, then `ifAllGranted`, and finally, `if AnyGranted`.

`AccessControlListTag` is used to include content if the current principal has an ACL to the indicated domain object.

The following JSP fragment illustrates how to use the `AccessControlListTag`:

```
<security:accesscontrollist domainObject="${contact}" hasPermission="8,16">
<td><a href="<c:url value="del.htm"><c:param name="contactId" value="${contact.id}"/></c:url>">Del</a></td>
</security:accesscontrollist>
```

This tag would cause the tag's body to be output if the principal holds either permission 16 or permission 1 for the "contact" domain object. The numbers are actually integers that are used with `BasePermission` bit masking. Please refer to the ACL section of this reference guide to understand more about the ACL capabilities of Spring Security.

`AclTag` is part of the old ACL module and should be considered deprecated. For the sake of historical reference, works exactly the samae as `AccessControlListTag`.

Chapter 23. Secure Object Implementations

23.1. AOP Alliance (MethodInvocation) Security Interceptor

Prior to Spring Security 2.0, securing `MethodInvocation`s needed quite a lot of boiler plate configuration. Now the recommended approach for method security is to use namespace configuration. This way the method security infrastructure beans are configured automatically for you so you don't really need to know about the implementation classes. We'll just provide a quick overview of the classes that are involved here.

Method security in enforced using a `MethodSecurityInterceptor`, which secures `MethodInvocation`s. Depending on the configuration approach, an interceptor may be specific to a single bean or shared between multiple beans. The interceptor uses a `MethodDefinitionSource` instance to obtain the configuration attributes that apply to a particular method invocation. `MapBasedMethodDefinitionSource` is used to store configuration attributes keyed by method names (which can be wildcarded) and will be used internally when the attributes are defined in the application context using the `<intercept-methods>` or `<protect-point>` elements. Other implementations will be used to handle annotation-based configuration.

23.1.1. Explicit MethodSecurityIterceptor Configuration

You can of course configure a `MethodSecurityIterceptor` directly in your application context for use with one of Spring AOP's proxying mechanisms:

```
<bean id="bankManagerSecurity"
    class="org.springframework.security.intercept.method.aopalliance.MethodSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="afterInvocationManager" ref="afterInvocationManager"/>
  <property name="objectDefinitionSource">
    <value>
      org.springframework.security.context.BankManager.delete*=ROLE_SUPERVISOR
      org.springframework.security.context.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISOR
    </value>
  </property>
</bean>
```

23.2. AspectJ (JoinPoint) Security Interceptor

The AspectJ security interceptor is very similar to the AOP Alliance security interceptor discussed in the previous section. Indeed we will only discuss the differences in this section.

The AspectJ interceptor is named `AspectJSecurityInterceptor`. Unlike the AOP Alliance security interceptor, which relies on the Spring application context to weave in the security interceptor via proxying, the `AspectJSecurityInterceptor` is weaved in via the AspectJ compiler. It would not be uncommon to use both types of security interceptors in the same application, with `AspectJSecurityInterceptor` being used for domain object instance security and the AOP Alliance `MethodSecurityInterceptor` being used for services layer security.

Let's first consider how the `AspectJSecurityInterceptor` is configured in the Spring application context:

```
<bean id="bankManagerSecurity"
        class="org.springframework.security.intercept.method.aspectj.AspectJSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
```

```
  <property name="afterInvocationManager" ref="afterInvocationManager"/>
  <property name="objectDefinitionSource">
    <value>
        org.springframework.security.context.BankManager.delete*=ROLE_SUPERVISOR
        org.springframework.security.context.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISOR
    </value>
</property>
</bean>
```

As you can see, aside from the class name, the `AspectJSecurityInterceptor` is exactly the same as the AOP Alliance security interceptor. Indeed the two interceptors can share the same `objectDefinitionSource`, as the `ObjectDefinitionSource` works with `java.lang.reflect.Methods` rather than an AOP library-specific class. Of course, your access decisions have access to the relevant AOP library-specific invocation (ie `MethodInvocation` or `JoinPoint`) and as such can consider a range of addition criteria when making access decisions (such as method arguments).

Next you'll need to define an AspectJ `aspect`. For example:

```
package org.springframework.security.samples.aspectj;

import org.springframework.security.intercept.method.aspectj.AspectJSecurityInterceptor;
import org.springframework.security.intercept.method.aspectj.AspectJCallback;
import org.springframework.beans.factory.InitializingBean;

public aspect DomainObjectInstanceSecurityAspect implements InitializingBean {

private AspectJSecurityInterceptor securityInterceptor;

pointcut domainObjectInstanceExecution(): target(PersistableEntity)
      && execution(public * *(..)) && !within(DomainObjectInstanceSecurityAspect);

Object around(): domainObjectInstanceExecution() {
  if (this.securityInterceptor == null) {
    return proceed();
  }

  AspectJCallback callback = new AspectJCallback() {
      public Object proceedWithObject() {
        return proceed();
      }
  };

  return this.securityInterceptor.invoke(thisJoinPoint, callback);
}

public AspectJSecurityInterceptor getSecurityInterceptor() {
  return securityInterceptor;
}

public void setSecurityInterceptor(AspectJSecurityInterceptor securityInterceptor) {
  this.securityInterceptor = securityInterceptor;
}

public void afterPropertiesSet() throws Exception {
  if (this.securityInterceptor == null)
    throw new IllegalArgumentException("securityInterceptor required");
  }
}
```

In the above example, the security interceptor will be applied to every instance of `PersistableEntity`, which is an abstract class not shown (you can use any other class or `pointcut` expression you like). For those curious, `AspectJCallback` is needed because the `proceed();` statement has special meaning only within an `around()` body. The `AspectJSecurityInterceptor` calls this anonymous `AspectJCallback` class when it wants the target object to continue.

You will need to configure Spring to load the aspect and wire it with the `AspectJSecurityInterceptor`. A

bean declaration which achieves this is shown below:

```
<bean id="domainObjectInstanceSecurityAspect"
     class="org.springframework.security.samples.aspectj.DomainObjectInstanceSecurityAspect"
     factory-method="aspectOf">
  <property name="securityInterceptor" ref="aspectJSecurityInterceptor"/>
</bean>
```

That's it! Now you can create your beans from anywhere within your application, using whatever means you think fit (eg `new Person();`) and they will have the security interceptor applied.

23.3. FilterInvocation Security Interceptor

To secure `FilterInvocations`, developers need to add a `FilterSecurityInterceptor` to their filter chain. A typical configuration example is provided below:

In the application context you will need to configure three beans:

```
<bean id="exceptionTranslationFilter"
        class="org.springframework.security.ui.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
</bean>

<bean id="authenticationEntryPoint"
        class="org.springframework.security.ui.webapp.AuthenticationProcessingFilterEntryPoint">
  <property name="loginFormUrl" value="/acegilogin.jsp"/>
  <property name="forceHttps" value="false"/>
</bean>

<bean id="filterSecurityInterceptor"
        class="org.springframework.security.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="objectDefinitionSource">
    <security:filter-invocation-definition-source>
      <security:intercept-url pattern="/secure/super/**" access="ROLE_WE_DONT_HAVE"/>
      <security:intercept-url pattern="/secure/**" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
    </security:filter-invocation-definition-source>
  </property>
</bean>
```

The `ExceptionTranslationFilter` provides the bridge between Java exceptions and HTTP responses. It is solely concerned with maintaining the user interface. This filter does not do any actual security enforcement. If an `AuthenticationException` is detected, the filter will call the AuthenticationEntryPoint to commence the authentication process (e.g. a user login).

The `AuthenticationEntryPoint` will be called if the user requests a secure HTTP resource but they are not authenticated. The class handles presenting the appropriate response to the user so that authentication can begin. Three concrete implementations are provided with Spring Security: `AuthenticationProcessingFilterEntryPoint` for commencing a form-based authentication, `BasicProcessingFilterEntryPoint` for commencing a HTTP Basic authentication process, and `CasProcessingFilterEntryPoint` for commencing a JA-SIG Central Authentication Service (CAS) login. The `AuthenticationProcessingFilterEntryPoint` and `CasProcessingFilterEntryPoint` have optional properties related to forcing the use of HTTPS, so please refer to the JavaDocs if you require this.

`FilterSecurityInterceptor` is responsible for handling the security of HTTP resources. Like any other security interceptor, it requires a reference to an `AuthenticationManager` and an `AccessDecisionManager`,

which are both discussed in separate sections below. The `FilterSecurityInterceptor` is also configured with configuration attributes that apply to different HTTP URL requests. A full discussion of configuration attributes is provided in the High Level Design section of this document.

The `FilterSecurityInterceptor` can be configured with configuration attributes in two ways. The first, which is shown above, is using the `<filter-invocation-definition-source>` namespace element. This is similar to the `<filter-chain-map>` used to configure a `FilterChainProxy` but the `<intercept-url>` child elements only use the `pattern` and `access` attributes. The second is by writing your own `ObjectDefinitionSource`, although this is beyond the scope of this document. Irrespective of the approach used, the `ObjectDefinitionSource` is responsible for returning a `ConfigAttributeDefinition` object that contains all of the configuration attributes associated with a single secure HTTP URL.

It should be noted that the `FilterSecurityInterceptor.setObjectDefinitionSource()` method actually expects an instance of `FilterInvocationDefinitionSource`. This is a marker interface which subclasses `ObjectDefinitionSource`. It simply denotes the `ObjectDefinitionSource` understands `FilterInvocations`. In the interests of simplicity we'll continue to refer to the `FilterInvocationDefinitionSource` as an `ObjectDefinitionSource`, as the distinction is of little relevance to most users of the `FilterSecurityInterceptor`.

When using the namespace option to configure the interceptor, commas are used to delimit the different configuration attributes that apply to each HTTP URL. Each configuration attribute is assigned into its own `SecurityConfig` object. The `SecurityConfig` object is discussed in the High Level Design section. The `ObjectDefinitionSource` created by the property editor, `FilterInvocationDefinitionSource`, matches configuration attributes against `FilterInvocations` based on expression evaluation of the request URL. Two standard expression syntaxes are supported. The default is to treat all expressions as Apache Ant paths and regular expressions are also supported for ore complex cases. The `path-type` attribute is used to specify the type of pattern being used. It is not possible to mix expression syntaxes within the same definition. For example, the previous configuration using regular expressions instead of Ant paths would be written as follows:

```
<bean id="filterInvocationInterceptor"
        class="org.springframework.security.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="runAsManager" ref="runAsManager"/>
  <property name="objectDefinitionSource">
    <security:filter-invocation-definition-source path-type="regex">
      <security:intercept-url pattern="\A/secure/super/.*\Z" access="ROLE_WE_DONT_HAVE"/>
      <security:intercept-url pattern="\A/secure/.*\" access="ROLE_SUPERVISOR,ROLE_TELLER"/>
    </security:filter-invocation-definition-source>
  </property>
</bean>
```

Irrespective of the type of expression syntax used, expressions are always evaluated in the order they are defined. Thus it is important that more specific expressions are defined higher in the list than less specific expressions. This is reflected in our example above, where the more specific `/secure/super/` pattern appears higher than the less specific `/secure/` pattern. If they were reversed, the `/secure/` pattern would always match and the `/secure/super/` pattern would never be evaluated.

As with other security interceptors, the `validateConfigAttributes` property is observed. When set to `true` (the default), at startup time the `FilterSecurityInterceptor` will evaluate if the provided configuration attributes are valid. It does this by checking each configuration attribute can be processed by either the `AccessDecisionManager` or the `RunAsManager`. If neither of these can process a given configuration attribute, an exception is thrown.

Chapter 24. Domain Object Security

24.1. Overview

PLEASE NOTE: Before release 2.0.0, Spring Security was known as Acegi Security. An ACL module was provided with the old Acegi Security releases under the `org.[acegisecurity/springsecurity].acl` package. This old package is now deprecated and will be removed in a future release of Spring Security. This chapter covers the new ACL module, which is officially recommended from Spring Security 2.0.0 and above, and can be found under the `org.springframework.security.acls` package.

Complex applications often will find the need to define access permissions not simply at a web request or method invocation level. Instead, security decisions need to comprise both who (`Authentication`), where (`MethodInvocation`) and what (`SomeDomainObject`). In other words, authorization decisions also need to consider the actual domain object instance subject of a method invocation.

Imagine you're designing an application for a pet clinic. There will be two main groups of users of your Spring-based application: staff of the pet clinic, as well as the pet clinic's customers. The staff will have access to all of the data, whilst your customers will only be able to see their own customer records. To make it a little more interesting, your customers can allow other users to see their customer records, such as their "puppy preschool" mentor or president of their local "Pony Club". Using Spring Security as the foundation, you have several approaches that can be used:

1. Write your business methods to enforce the security. You could consult a collection within the `Customer` domain object instance to determine which users have access. By using the `SecurityContextHolder.getContext().getAuthentication()`, you'll be able to access the `Authentication` object.

2. Write an `AccessDecisionVoter` to enforce the security from the `GrantedAuthority[]`s stored in the `Authentication` object. This would mean your `AuthenticationManager` would need to populate the `Authentication` with custom `GrantedAuthority`[]s representing each of the `Customer` domain object instances the principal has access to.

3. Write an `AccessDecisionVoter` to enforce the security and open the target `Customer` domain object directly. This would mean your voter needs access to a DAO that allows it to retrieve the `Customer` object. It would then access the `Customer` object's collection of approved users and make the appropriate decision.

Each one of these approaches is perfectly legitimate. However, the first couples your authorization checking to your business code. The main problems with this include the enhanced difficulty of unit testing and the fact it would be more difficult to reuse the `Customer` authorization logic elsewhere. Obtaining the `GrantedAuthority[]`s from the `Authentication` object is also fine, but will not scale to large numbers of `Customers`. If a user might be able to access 5,000 `Customers` (unlikely in this case, but imagine if it were a popular vet for a large Pony Club!) the amount of memory consumed and time required to construct the `Authentication` object would be undesirable. The final method, opening the `Customer` directly from external code, is probably the best of the three. It achieves separation of concerns, and doesn't misuse memory or CPU cycles, but it is still inefficient in that both the `AccessDecisionVoter` and the eventual business method itself will perform a call to the DAO responsible for retrieving the `Customer` object. Two accesses per method invocation is clearly undesirable. In addition, with every approach listed you'll need to write your own access control list (ACL) persistence and business logic from scratch.

Fortunately, there is another alternative, which we'll talk about below.

24.2. Key Concepts

Spring Security's ACL services are shipped in the `spring-security-acl-xxx.jar`. You will need to add this JAR to your classpath to use Spring Security's domain object instance security capabilities.

Spring Security's domain object instance security capabilities centre on the concept of an access control list (ACL). Every domain object instance in your system has its own ACL, and the ACL records details of who can and can't work with that domain object. With this in mind, Spring Security delivers three main ACL-related capabilities to your application:

- A way of efficiently retrieving ACL entries for all of your domain objects (and modifying those ACLs)
- A way of ensuring a given principal is permitted to work with your objects, before methods are called
- A way of ensuring a given principal is permitted to work with your objects (or something they return), after methods are called

As indicated by the first bullet point, one of the main capabilities of the Spring Security ACL module is providing a high-performance way of retrieving ACLs. This ACL repository capability is extremely important, because every domain object instance in your system might have several access control entries, and each ACL might inherit from other ACLs in a tree-like structure (this is supported out-of-the-box by Spring Security, and is very commonly used). Spring Security's ACL capability has been carefully designed to provide high performance retrieval of ACLs, together with pluggable caching, deadlock-minimizing database updates, independence from ORM frameworks (we use JDBC directly), proper encapsulation, and transparent database updating.

Given databases are central to the operation of the ACL module, let's explore the four main tables used by default in the implementation. The tables are presented below in order of size in a typical Spring Security ACL deployment, with the table with the most rows listed last:

- ACL_SID allows us to uniquely identify any principal or authority in the system ("SID" stands for "security identity"). The only columns are the ID, a textual representation of the SID, and a flag to indicate whether the textual representation refers to a prncipal name or a `GrantedAuthority`. Thus, there is a single row for each unique principal or `GrantedAuthority`. When used in the context of receiving a permission, a SID is generally called a "recipient".
- ACL_CLASS allows us to uniquely identify any domain object class in the system. The only columns are the ID and the Java class name. Thus, there is a single row for each unique Class we wish to store ACL permissions for.
- ACL_OBJECT_IDENTITY stores information for each unique domain object instance in the system. Columns include the ID, a foreign key to the ACL_CLASS table, a unique identifier so we know which ACL_CLASS instance we're providing information for, the parent, a foreign key to the ACL_SID table to represent the owner of the domain object instance, and whether we allow ACL entries to inherit from any parent ACL. We have a single row for every domain object instance we're storing ACL permissions for.
- Finally, ACL_ENTRY stores the individual permissions assigned to each recipient. Columns include a foreign key to the ACL_OBJECT_IDENTITY, the recipient (ie a foreign key to ACL_SID), whether we'll be auditing or not, and the integer bit mask that represents the actual permission being granted or denied. We have a single row for every recipient that receives a permission to work with a domain object.

As mentioned in the last paragraph, the ACL system uses integer bit masking. Don't worry, you need not be aware of the finer points of bit shifting to use the ACL system, but suffice to say that we have 32 bits we can switch on or off. Each of these bits represents a permission, and by default the permissions are read (bit 0), write (bit 1), create (bit 2), delete (bit 3) and administer (bit 4). It's easy to implement your own `Permission` instance if you wish to use other permissions, and the remainder of the ACL framework will operate without knowledge of your extensions.

It is important to understand that the number of domain objects in your system has absolutely no bearing on the fact we've chosen to use integer bit masking. Whilst you have 32 bits available for permissions, you could have billions of domain object instances (which will mean billions of rows in ACL_OBJECT_IDENTITY and quite probably ACL_ENTRY). We make this point because we've found sometimes people mistakenly believe they need a bit for each potential domain object, which is not the case.

Now that we've provided a basic overview of what the ACL system does, and what it looks like at a table structure, let's explore the key interfaces. The key interfaces are:

- `Acl`: Every domain object has one and only one `Acl` object, which internally holds the `AccessControlEntry`s as well as knows the owner of the `Acl`. An Acl does not refer directly to the domain object, but instead to an `ObjectIdentity`. The `Acl` is stored in the ACL_OBJECT_IDENTITY table.
- `AccessControlEntry`: An `Acl` holds multiple `AccessControlEntry`s, which are often abbreviated as ACEs in the framework. Each ACE refers to a specific tuple of `Permission`, `Sid` and `Acl`. An ACE can also be granting or non-granting and contain audit settings. The ACE is stored in the ACL_ENTRY table.
- `Permission`: A permission represents a particular immutable bit mask, and offers convenience functions for bit masking and outputting information. The basic permissions presented above (bits 0 through 4) are contained in the `BasePermission` class.
- `Sid`: The ACL module needs to refer to principals and `GrantedAuthority[]`s. A level of indirection is provided by the `Sid` interface, which is an abbreviation of "security identity". Common classes include `PrincipalSid` (to represent the principal inside an `Authentication` object) and `GrantedAuthoritySid`. The security identity information is stored in the ACL_SID table.
- `ObjectIdentity`: Each domain object is represented internally within the ACL module by an `ObjectIdentity`. The default implementation is called `ObjectIdentityImpl`.
- `AclService`: Retrieves the `Acl` applicable for a given `ObjectIdentity`. In the included implementation (`JdbcAclService`), retrieval operations are delegated to a `LookupStrategy`. The `LookupStrategy` provides a highly optimized strategy for retrieving ACL information, using batched retrievals (`BasicLookupStrategy`) and supporting custom implementations that leverage materialized views, hierarchical queries and similar performance-centric, non-ANSI SQL capabilities.
- `MutableAclService`: Allows a modified `Acl` to be presented for persistence. It is not essential to use this interface if you do not wish.

Please note that our out-of-the-box AclService and related database classes all use ANSI SQL. This should therefore work with all major databases. At the time of writing, the system had been successfully tested using Hypersonic SQL, PostgreSQL, Microsoft SQL Server and Oracle.

Two samples ship with Spring Security that demonstrate the ACL module. The first is the Contacts Sample, and the other is the Document Management System (DMS) Sample. We suggest taking a look over these for examples.

24.3. Getting Started

To get starting using Spring Security's ACL capability, you will need to store your ACL information somewhere. This necessitates the instantiation of a `DataSource` using Spring. The `DataSource` is then injected into a `JdbcMutableAclService` and `BasicLookupStrategy` instance. The latter provides high-performance

ACL retrieval capabilities, and the former provides mutator capabilities. Refer to one of the samples that ship with Spring Security for an example configuration. You'll also need to populate the database with the four ACL-specific tables listed in the last section (refer to the ACL samples for the appropriate SQL statements).

Once you've created the required schema and instantiated `JdbcMutableAclService`, you'll next need to ensure your domain model supports interoperability with the Spring Security ACL package. Hopefully `ObjectIdentityImpl` will prove sufficient, as it provides a large number of ways in which it can be used. Most people will have domain objects that contain a `public Serializable getId()` method. If the return type is long, or compatible with long (eg an int), you will find you need not give further consideration to `ObjectIdentity` issues. Many parts of the ACL module rely on long identifiers. If you're not using long (or an int, byte etc), there is a very good chance you'll need to reimplement a number of classes. We do not intend to support non-long identifiers in Spring Security's ACL module, as longs are already compatible with all database sequences, the most common identifier data type, and are of sufficient length to accommodate all common usage scenarios.

The following fragment of code shows how to create an `Acl`, or modify an existing `Acl`:

```
// Prepare the information we'd like in our access control entry (ACE)
ObjectIdentity oi = new ObjectIdentityImpl(Foo.class, new Long(44));
Sid sid = new PrincipalSid("Samantha");
Permission p = BasePermission.ADMINISTRATION;

// Create or update the relevant ACL
MutableAcl acl = null;
try {
  acl = (MutableAcl) aclService.readAclById(oi);
} catch (NotFoundException nfe) {
  acl = aclService.createAcl(oi);
}

// Now grant some permissions via an access control entry (ACE)
acl.insertAce(acl.getEntries().length, p, sid, true);
aclService.updateAcl(acl);
```

In the example above, we're retrieving the ACL associated with the "Foo" domain object with identifier number 44. We're then adding an ACE so that a principal named "Samantha" can "administer" the object. The code fragment is relatively self-explanatory, except the insertAce method. The first argument to the insertAce method is determining at what position in the Acl the new entry will be inserted. In the example above, we're just putting the new ACE at the end of the existing ACEs. The final argument is a boolean indicating whether the ACE is granting or denying. Most of the time it will be granting (true), but if it is denying (false), the permissions are effectively being blocked.

Spring Security does not provide any special integration to automatically create, update or delete ACLs as part of your DAO or repository operations. Instead, you will need to write code like shown above for your individual domain objects. It's worth considering using AOP on your services layer to automatically integrate the ACL information with your services layer operations. We've found this quite an effective approach in the past.

Once you've used the above techniques to store some ACL information in the database, the next step is to actually use the ACL information as part of authorization decision logic. You have a number of choices here. You could write your own `AccessDecisionVoter` or `AfterInvocationProvider` that respectively fires before or after a method invocation. Such classes would use `AclService` to retrieve the relevant ACL and then call `Acl.isGranted(Permission[] permission, Sid[] sids, boolean administrativeMode)` to decide whether permission is granted or denied. Alternately, you could use our `AclEntryVoter`, `AclEntryAfterInvocationProvider` or `AclEntryAfterInvocationCollectionFilteringProvider` classes. All of these classes provide a declarative-based approach to evaluating ACL information at runtime, freeing you from needing to write any code. Please refer to the sample applications to learn how to use these classes.

Appendix A. Security Database Schema

There are various database schema used by the framework and this appendix provides a single reference point to them all. You only need to provide the tables for the areas of functonality you require.

DDL statements are given for the HSQLDB database. You can use these as a guideline for defining the schema for the database you are using.

A.1. User Schema

The standard JDBC implementation of the `UserDetailsService` requires tables to load the password, account status (enabled or disabled) and a list of authorities (roles) for the user.

```
  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,
      authority varchar_ignorecase(50) not null,
      constraint fk_authorities_users foreign key(username) references users(username));
      create unique index ix_auth_username on authorities (username,authority);;
```

A.1.1. Group Authorities

Spring Security 2.0 introduced support for group authorities

```
create table groups (
  id bigint generated by default as identity(start with 0) primary key,
  group_name varchar_ignorecase(50) not null);

create table group_authorities (
  group_id bigint not null,
  authority varchar(50) not null,
  constraint fk_group_authorities_group foreign key(group_id) references groups(id));

create table group_members (
  id bigint generated by default as identity(start with 0) primary key,
  username varchar(50) not null,
  group_id bigint not null,
  constraint fk_group_members_group foreign key(group_id) references groups(id));
```

A.2. Persistent Login (Remember-Me) Schema

This table is used to store data used by the more secure persistent token remember-me implementation. If you are using `JdbcTokenRepositoryImpl` either directly or through the namespace, then you will need this table.

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

A.3. ACL Schema

The tables used by the Spring Security ACL implementation.

```
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) );
```

Appendix B. The Security Namespace

This appendix provides a reference to the elements available in the security namespace and information on the underlying beans they create (a knowledge of the individual classes and how they work together is assumed - you can find more information in the project Javadoc and elsewhere in this document). If you haven't used the namespace before, please read the introductory chapter on namespace configuration, as this is intended as a supplement to the information there. Using a good quality XML editor while editing a configuration based on the schema is recommended as this will provide contextual information on which elements and attributes are available as well as comments explaining their purpose.

B.1. Web Application Security - the `<http>` Element

The `<http>` element encapsulates the security configuration for the web layer of your application. It creates a `FilterChainProxy` bean named "springSecurityFilterChain" which maintains the stack of security filters which make up the web security configuration [1]. Some core filters are always created and others will be added to the stack depending on the attributes child elements which are present. The positions of the standard filters are fixed (see the filter order table in the namespace introduction), removing a common source of errors with previous versions of the framework when users had to configure the filter chain explicitly in the`FilterChainProxy` bean. You can, of course, still do this if you need full control of the configuration.

All filters which require a reference to the `AuthenticationManager` will be automatically injected with the internal instance created by the namespace configuration (see the introductory chapter for more on the `AuthenticationManager`).

The `<http>` namespace block always creates an `HttpSessionContextIntegrationFilter`, an `ExceptionTranslationFilter` and a `FilterSecurityInterceptor`. These are fixed and cannot be replaced with alternatives.

B.1.1. `<http>` Attributes

The attributes on the `<http>` element control some of the properties on the core filters.

B.1.1.1. `servlet-api-provision`

Provides versions of `HttpServletRequest` security methods such as `isUserInRole()` and `getPrincipal()` which are implemented by adding a `SecurityContextHolderAwareRequestFilter` bean to the stack. Defaults to "true".

B.1.1.2. `path-type`

Controls whether URL patterns are interpreted as ant paths (the default) or regular expressions. In practice this sets a particular `UrlMatcher` instance on the `FilterChainProxy`.

B.1.1.3. `lowercase-comparisons`

Whether test URLs should be converted to lower case prior to comparing with defined path patterns. If unspecified, defaults to "true"

[1] See the introductory chapter for how to set up the mapping from your `web.xml`

B.1.1.4. `session-fixation-protection`

Indicates whether an existing session should be invalidated when a user authenticates and a new session started. If set to "none" no change will be made. "newSession" will create a new empty session. "migrateSession" will create a new session and copy the session attributes to the new session. Defaults to "migrateSession".

If enabled this will add a `SessionFixationProtectionFilter` to the stack. The session fixation protection options on namespace-created instances of `AbstractProcessingFilter` will also be set appropriately.

B.1.1.5. `realm`

Sets the realm name used for basic authentication (if enabled). Corresponds to the `realmName` proerty on `BasicProcessingFilterEntryPoint`.

B.1.1.6. `entry-point-ref`

Normally the `AuthenticationEntryPoint` used will be set depending on which authentication mechanisms have been configured. This attribute allows this behaviour to be overridden by defining a customized `AuthenticationEntryPoint` bean which will start the authentication process.

B.1.1.7. `access-decision-manager-ref`

Optional attribute specifying the ID of the `AccessDecisionManager` implementation which should be used for authorizing HTTP requests. By default an `AffirmativeBased` implementation is used for with a `RoleVoter` and an `AuthenticatedVoter`.

B.1.1.8. `access-denied-page`

Allows the access denied page to be set (the user will be redirected here if an `AccessDeniedException` is raised). Corresponds to the `errorPage` property set on the `AccessDeniedHandlerImpl` which is used by the `ExceptionTranslationFilter`.

B.1.1.9. `once-per-request`

Corresponds to the `observeOncePerRequest` property of `FilterSecurityInterceptor`. Defaults to "true".

B.1.1.10. `create-session`

Controls the eagerness with which an HTTP session is created. If not set, defaults to "ifRequired". Other options are "always" and "never". The setting of this attribute affect the `allowSessionCreation` and `forceEagerSessionCreation` properties of `HttpSessionContextIntegrationFilter`. `allowSessionCreation` will always be true unless this attribute is set to "never". `forceEagerSessionCreation` is "false" unless it is set to "always". So the default configuration allows session creation but does not force it. The exception is if concurrent session control is enabled, when `forceEagerSessionCreation` will be set to true, regardless of what the setting is here. Using "never" would then cause an exception during the initialization of `HttpSessionContextIntegrationFilter`.

B.1.2. The `<intercept-url>` Element

This element is used to define the set of URL patterns that the application is interested in and to configure how they should be handled. It is used to construct the `FilterInvocationDefinitionSource` used by the `FilterSecurityInterceptor` and to exclude particular patterns from the filter chain entirely (by setting the

attribute `filters="none"`). It is also responsible for configuring a `ChannelProcessingFilter` if particular URLs need to be accessed by HTTPS, for example.

B.1.2.1. `pattern`

The pattern which defines the URL path. The content will depend on the `path-type` attribute from the containing http element, so will default to ant path syntax.

B.1.2.2. `method`

The HTTP Method which will be used in combination with the pattern to match an incoming request. If omitted, any method will match.

B.1.2.3. `access`

Lists the access attributes which will be stored in the `FilterInvocationDefinitionSource` for the defined URL pattern/method combination. This should be a comma-separated list of the attributes (such as role names).

B.1.2.4. `requires-channel`

Can be "http" or "https" depending on whether a particular URL pattern should be accessed over HTTP or HTTPS respectively. Alternatively the value "any" can be used when there is no preference. If this attribute is present on any `<intercept-url>` element, then a `ChannelProcessingFilter` will be added to the filter stack and its additional dependencies added to the application context. See the chapter on channel security for an example configuration using traditional beans.

If a `<port-mappings>` configuration is added, this will be used to by the `SecureChannelProcessor` and `InsecureChannelProcessor` beans to determine the ports used for redirecting to HTTP/HTTPS.

B.1.3. The `<port-mappings>` Element

By default, an instance of `PortMapperImpl` will be added to the configuration for use in redirecting to secure and insecure URLs. This element can optionally be used to override the default mappings which that class defines. Each child `<port-mapping>` element defines a pair of HTTP:HTTPS ports. The default mappings are 80:443 and 8080:8443. An example of overriding these can be found in the namespace introduction.

B.1.4. The `<form-login>` Element

Used to add an `AuthenticationProcessingFilter` to the filter stack and an `AuthenticationProcessingFilterEntryPoint` to the application context to provide authentication on demand. This will always take precedence over other namespace-created entry points. If no attributes are supplied, a login page will be generated automatically at the URL "/spring-security-login" [2] The behaviour can be customized using the following attributes.

B.1.4.1. `login-page`

The URL that should be used to render the login page. Maps to the `loginFormUrl` property of the `AuthenticationProcessingFilterEntryPoint`. Defaults to "/spring-security-login".

[2]This feature is really just provided for convenience and is not intended for production (where a view technology will have been chosen and can be used to render a customized login page). The class `DefaultLoginPageGeneratingFilter` is responsible for rendering the login page and will provide login forms for both normal form login and/or OpenID if required.

B.1.4.2. `login-processing-url`

Maps to the `filterProcessesUrl` property of `AuthenticationProcessingFilter`. The default value is "/j_spring_security_check".

B.1.4.3. `default-target-url`

Maps to the `defaultTargetUrl` property of `AuthenticationProcessingFilter`. If not set, the default value is "/" (the application root). A user will be taken to this URL after logging in, provided they were not asked to login while attempting to access a secured resource, when they will be taken to the originally requested URL.

B.1.4.4. `always-use-default-target`

If set to "true", the user will always start at the value given by `default-target-url`, regardless of how they arrived at the login page. Maps to the `alwaysUseDefaultTargetUrl` property of `AuthenticationProcessingFilter`. Default value is "false".

B.1.4.5. `authentication-failure-url`

Maps to the `authenticationFailureUrl` property of `AuthenticationProcessingFilter`. Defines the URL the browser will be redirected to on login failure. Defaults to "/spring_security_login?login_error", which will be automatically handled by the automatic login page generator, re-rendering the login page with an error message.

B.1.5. The `<http-basic>` Element

Adds a `BasicProcessingFilter` and `BasicProcessingFilterEntryPoint` to the configuration. The latter will only be used as the configuration entry point if form-based login is not enabled.

B.1.6. The `<remember-me>` Element

Adds the `RememberMeProcessingFilter` to the stack. This in turn will be configured with either a `TokenBasedRememberMeServices`, a `PersistentTokenBasedRememberMeServices` or a user-specified bean implementing `RememberMeServices` depending on the attribute settings.

B.1.6.1. `data-source-ref`

If this is set, `PersistentTokenBasedRememberMeServices` will be used and configured with a `JdbcTokenRepositoryImpl` instance.

B.1.6.2. `token-repository-ref`

Configures a `PersistentTokenBasedRememberMeServices` but allows the use of a custom `PersistentTokenRepository` bean.

B.1.6.3. `services-ref`

Allows complete control of the `RememberMeServices` implementation that will be used by the filter. The value should be the Id of a bean in the application context which implements this interface.

B.1.6.4. `token-repository-ref`

Configures a `PersistentTokenBasedRememberMeServices` but allows the use of a custom `PersistentTokenRepository` bean.

B.1.6.5. The `key` Attribute

Maps to the "key" property of `AbstractRememberMeServices`. Should be set to a unique value to ensure that remember-me cookies are only valid within the one application [3].

B.1.6.6. `token-validity-seconds`

Maps to the `tokenValiditySeconds` property of `AbstractRememberMeServices`. Specifies the period in seconds for which the remember-me cookie should be valid. By default it will be valid for 14 days.

B.1.6.7. `user-service-ref`

The remember-me services implementations require access to a `UserDetailsService`, so there has to be one defined in the application context. If there is only one, it will be selected and used automatically by the namespace configuration. If there are multiple instances, you can specify a bean Id explicitly using this attribute.

B.1.7. The `<concurrent-session-control>` Element

Adds support for concurrent session control, allowing limits to be placed on the number of active sessions a user can have. A `ConcurrentSessionFilter` will be created, along with a `ConcurrentSessionControllerImpl` and an instance of `SessionRegistry` (a `SessionRegistryImpl` instance unless the user wishes to use a custom bean). The controller is registered with the namespace's `AuthenticationManager` (`ProviderManager`). Other namespace-created beans which require a reference to the `SessionRegistry` will automatically have it injected.

Note that the `forceEagerSessionCreation` of `HttpSessionContextIntegrationFilter` will be set to `true` if concurrent session control is in use.

B.1.7.1. The `max-sessions` attribute

Maps to the `maximumSessions` property of `ConcurrentSessionControllerImpl`.

B.1.7.2. The `expired-url` attribute

The URL a user will be redirected to if they attempt to use a session which has been "expired" by the concurrent session controller because the user has exceeded the number of allowed sessions and has logged in again elsewhere. Should be set unless `exception-if-maximum-exceeded` is set. If no value is supplied, an expiry message will just be written directly back to the response.

B.1.7.3. The `exception-if-maximum-exceeded` attribute

If set to "true" a `ConcurrentLoginException` should be raised when a user attempts to exceed the maximum allowed number of sessions. The default behaviour is to expire the original session.

B.1.7.4. The `session-registry-alias` and `session-registry-ref` attributes

The user can supply their own `SessionRegistry` implementation using the `session-registry-ref` attribute.

[3] This doesn't affect the use of `PersistentTokenBasedRememberMeServices`, where the tokens are stored on the server side.

The other concurrent session control beans will be wired up to use it.

It can also be useful to have a reference to the internal session registry for use in your own beans or an admin interface. You can expose the interal bean using the `session-registry-alias` attribute, giving it a name that you can use elsewhere in your configuration.

B.1.8. The `<anonymous>` Element

Adds an `AnonymousProcessingFilter` to the stack and an `AnonymousAuthenticationProvider`. Required if you are using the `IS_AUTHENTICATED_ANONYMOUSLY` attribute.

B.1.9. The `<x509>` Element

Adds support for X.509 authentication. An `X509PreAuthenticatedProcessingFilter` will be added to the stack and a `PreAuthenticatedProcessingFilterEntryPoint` bean will be created. The latter will only be used if no other authentication mechanisms are in use (it's only functionality is to return an HTTP 403 error code). A `PreAuthenticatedAuthenticationProvider` will also be created which delegates the loading of user authorities to a `UserDetailsService`.

B.1.9.1. The `subject-principal-regex` attribute

Defines a regular expression which will be used to extract the username from the certificate (for use with the `UserDetailsService`).

B.1.9.2. The `user-service-ref` attribute

Allows a specific `UserDetailsService` to be used with X.509 in the case where multiple instances are configured. If not set, an attempt will be made to locate a suitable instance automatically and use that.

B.1.10. The `<openid-login>` Element

Similar to `<form-login>` and has the same attributes. The default value for `login-processing-url` is "/j_spring_openid_security_check". An `OpenIDAuthenticationProcessingFilter` and `OpenIDAuthenticationProvider` will be registered. The latter requires a reference to a `UserDetailsService`. Again, this can be specified by Id, using the `user-service-ref` attribute, or will be located automatically in the application context.

B.1.11. The `<logout>` Element

Adds a `LogoutFilter` to the filter stack. This is configured with a `SecurityContextLogoutHandler`.

B.1.11.1. The `logout-url` attribute

The URL which will cause a logout (i.e. which will be processed by the filter). Defaults to "/j_spring_security_logout".

B.1.11.2. The `logout-success-url` attribute

The destination URL which the user will be taken to after logging out. Defaults to "/".

B.1.11.3. The `invalidate-session` attribute

Maps to the `invalidateHttpSession` of the `SecurityContextLogoutHandler`. Defaults to "true", so the session will be invalidated on logout.

B.2. Authentication Services

If you are using the namespace, an `AuthenticationManager` is automatically registered and will be used by all the namespace-created beans which need to reference it. The bean is an instance of Spring Security's `ProviderManager` class, which needs to be configured with a list of one or more `AuthenticationProvider` instances. These can either be created using syntax elements provided by the namespace, or they can be standard bean definitions, marked for addition to the list using the `custom-authentication-provider` element.

B.2.1. The <authentication-provider> Element

This element is basically a shorthand syntax for configuring a `DaoAuthenticationProvider`. `DaoAuthenticationProvider` loads user information from a `UserDetailsService` and compares the username/password combination with the values supplied at login. The `UserDetailsService` instance can be defined either by using an available namespace element (`jdbc-user-service` or by using the `user-service-ref` attribute to point to a bean defined elsewhere in the application context). You can find examples of these variations in the namespace introduction.

B.2.2. Using `<custom-authentication-provider>` to register an AuthenticationProvider

If you have written your own `AuthenticationProvider` implementation (or want to configure one of Spring Security's own implementations as a traditional bean for some reason, then you can use the following syntax to add it to the internal `ProviderManager`'s list:

```
<bean id="myAuthenticationProvider" class="com.something.MyAuthenticationProvider">
  <security:custom-authentication-provider />
</bean>
```

B.2.3. The `<authentication-manager>` Element

Since the `AuthenticationManager` will be automatically registered in the application context, this element is entirely optional. It allows you to define an alias name for the internal instance for use in your own configuration and also to supply a link to a `ConcurrentSessionController` if you are configuring concurrent session control yourself rather than through the namespace (a rare requirement). Its use is described in the namespace introduction.

B.3. Method Security

B.3.1. The `<global-method-security>` Element

This element is the primary means of adding support for securing methods on Spring Security beans. Methods can be secured by the use of annotations (defined at the interface or class level) or by defining a set of pointcuts as child elements, using AspectJ syntax.

Method security uses the same `AccessDecisionManager` configuration as web security, but this can be overridden as explained above Section B.1.1.7, “`access-decision-manager-ref`”, using the same attribute.

B.3.1.1. The `<secured-annotations>` and `<jsr250-annotations>` Attributes

Setting these to "true" will enable support for Spring Security's own `@Secured` annotations and JSR-250 annotations, respectively. They are both disabled by default. Use of JSR-250 annotations also adds a `Jsr250Voter` to the `AccessDecisionManager`, so you need to make sure you do this if you are using a custom implementation and want to use these annotations.

B.3.1.2. Securing Methods using `<protect-pointcut>`

Rather than defining security attributes on an individual method or class basis using the `@Secured` annotation, you can define cross-cutting security constraints across whole sets of methods and interfaces in your service layer using the `<protect-pointcut>` element. This has two attributes:

- `expression` - the pointcut expression
- `access` - the security attributes which apply

You can find an example in the namespace introduction.

B.3.2. LDAP Namespace Options

LDAP is covered in some details in its own chapter. We will expand on that here with some explanation of how the namespace options map to Spring beans. The LDAP implementation uses Spring LDAP extensively, so some familiarity with that project's API may be useful.

B.3.2.1. Defining the LDAP Server using the `<ldap-server>` Element

This element sets up a Spring LDAP `ContextSource` for use by the other LDAP beans, defining the location of the LDAP server and other information (such as a username and password, if it doesn't allow anonymous access) for connecting to it. It can also be used to create an embedded server for testing. Details of the syntax for both options are covered in the LDAP chapter. The actual `ContextSource` implementation is `DefaultSpringSecurityContextSource` which extends Spring LDAP's `LdapContextSource` class. The `manager-dn` and `manager-password` attributes map to the latter's `userDn` and `password` properties respectively.

If you only have one server defined in your application context, the other LDAP namespace-defined beans will use it automatically. Otherwise, you can give the element an "id" attribute and refer to it from other namespace beans using the `server-ref` attribute. This is actually the bean Id of the `ContextSource` instance, if you want to use it in other traditional Spring beans.

B.3.2.2. The `<ldap-provider>` Element

This element is shorthand for the creation of an `LdapAuthenticationProvider` instance. By default this will be configured with a `BindAuthenticator` instance and a `DefaultAuthoritiesPopulator`.

B.3.2.2.1. The `user-dn-pattern` Attribute

If your users are at a fixed location in the directory (i.e. you can work out the DN directly from the username without doing a directory search), you can use this attribute to map directly to the DN. It maps directly to the `userDnPatterns` property of `AbstractLdapAuthenticator`.

B.3.2.2.2. The `user-search-base` and `user-search-filter` Attributes

If you need to perform a search to locate the user in the directory, then you can set these attributes to control the search. The `BindAuthenticator` will be configured with a `FilterBasedLdapUserSearch` and the attribute values map directly to the first two arguments of that bean's constructor. If these attributes aren't set and no `user-dn-pattern` has been supplied as an alternative, then the default search values of `user-search-filter="(uid={0})"` and `user-search-base=""` will be used.

B.3.2.2.3. `group-search-filter`, `group-search-base`, `group-role-attribute` and `role-prefix` Attributes

The value of `group-search-base` is mapped to the `groupSearchBase` constructor argument of `DefaultAuthoritiesPopulator` and defaults to "ou=groups". The default filter value is "(uniqueMember={0})", which assumes that the entry is of type "groupOfUniqueNames". `group-role-attribute` maps to the `groupRoleAttribute` attribute and defaults to "cn". Similarly `role-prefix` maps to `rolePrefix` and defaults to "ROLE_".

B.3.2.2.4. The `<password-compare>` Element

This is used as child element to `<ldap-provider>` and switches the authentication strategy from `BindAuthenticator` to `PasswordComparisonAuthenticator`. This can optionally be supplied with a `hash` attribute or with a child `<password-encoder>` element to hash the password before submitting it to the directory for comparison.

B.3.2.3. The `<ldap-user-service>` Element

This element configures an LDAP `UserDetailsService`. The class used is `LdapUserDetailsService` which is a combination of a `FilterBasedLdapUserSearch` and a `DefaultAuthoritiesPopulator`. The attributes it supports have the same usage as in `<ldap-provider>`.