Spring Security

Reference Documentation

Ben Alex
Luke Taylor

Spring Security: Reference Documentation

by Ben Alex and Luke Taylor

3.0.8.RELEASE

Table of Contents

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 how Spring Security works, and some of the classes you might need to use, you should then read Part II, “Architecture and Implementation”. 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, which you need to understand if you want to do any serious customization. In this part, we'll introduce Spring Security 3.0, 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 have 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. Please also check out the project website [http://static.springsource.org/spring-security/site/index.html] as it has useful information on building the project, plus links to articles, videos and tutorials.

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, customisable security features.

As you probably know two major areas of application security are “authentication” and “authorization” (or “access-control”). These are the two main areas that Spring Security targets. “Authentication” 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 within 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
- Authentication based on pre-established request headers (such as 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)
- JEE container autentication (so you can still use Container Managed Authentication if desired)
- Kerberos
- 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 Path *
- Atlassian Crowd *
- Your own authentication systems (see below)

(* Denotes provided by a third party; check our integration page [http://acegisecurity.org/powering.html] 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. 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 whether 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 compared with the size today!), 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 became clear. 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 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 Spring Security

You can get hold of Spring Security in several ways. You can download a packaged distribution from the main Spring download page [http://www.springsource.com/download/community?project=Spring%20Security], download individual jars (and sample WAR files) from the Maven Central repository (or a SpringSource Maven repository for snapshot and milestone releases) or, alternatively, you can build the project from source yourself. See the project web site for more details.

Project Modules

In Spring Security 3.0, the codebase has been sub-divided into separate jars which more clearly separate different functionaltiy areas and third-party dependencies. If you are using Maven to build your project, then these are the modules you will add to your `pom.xml`. Even if you're not using Maven, we'd recommend that you consult the `pom.xml` files to get an idea of third-party dependencies and versions. Alternatively, a good idea is to examine the libraries that are included in the sample applications.

Core - `spring-security-core.jar`

Contains core authentication and access-contol classes and interfaces, remoting support and basic provisioning APIs. Required by any application which uses Spring Security. Supports standalone applications, remote clients, method (service layer) security and JDBC user provisioning. Contains the top-level packages:

- `org.springframework.security.core`
- `org.springframework.security.access`
- `org.springframework.security.authentication`
- `org.springframework.security.provisioning`
- `org.springframework.security.remoting`

Web - `spring-security-web.jar`

Contains filters and related web-security infrastructure code. Anything with a servlet API dependency. You'll need it if you require Spring Security web authentication services and URL-based access-control. The main package is `org.springframework.security.web`.

Config - `spring-security-config.jar`

Contains the security namespace parsing code (and hence nothing that you are likely yo use directly in your application). You need it if you are using the Spring Security XML namespace for configuration. The main package is `org.springframework.security.config`.

LDAP - `spring-security-ldap.jar`

LDAP authentication and provisioning code. Required if you need to use LDAP authentication or manage LDAP user entries. The top-level package is `org.springframework.security.ldap`.

ACL - `spring-security-acl.jar`

Specialized domain object ACL implementation. Used to apply security to specific domain object instances within your application. The top-level package is `org.springframework.security.acls`.

CAS - `spring-security-cas-client.jar`

Spring Security's CAS client integration. If you want to use Spring Security web authentication with a CAS single sign-on server. The top-level package is `org.springframework.security.cas`.

OpenID - `spring-security-openid.jar`

OpenID web authentication support. Used to authenticate users against an external OpenID server. `org.springframework.security.openid`. Requires OpenID4Java.

Checking out the Source

Since Spring Security is an Open Source project, we'd strongly encourage you to check out the source code using git. 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 git command:

```
git clone git://git.springsource.org/spring-security/spring-security.git
```

You can checkout specific versions from `https://src.springframework.org/svn/spring-security/tags/`.

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 [http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/htmlsingle/spring-framework-reference.htm]. 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 create 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 [http://www.springsource.com/products/sts] as it has special features for working with standard Spring namespaces.

To start using the security namespace in your application context, you first need to make sure that the `spring-security-config` jar is on your classpath. Then 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/spring-beans-3.0.xsd
          http://www.springframework.org/schema/security
          http://www.springframework.org/schema/security/spring-security-3.0.3.xsd">
    ...
</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 content 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

```
```

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

```
<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/spring-beans-3.0.xsd
            http://www.springframework.org/schema/security
            http://www.springframework.org/schema/security/spring-security-3.0.3.xsd">
    ...
</beans:beans>
```

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

Design of the Namespace

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 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.
- *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 to configure these 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 repository. In later sections we'll introduce more advanced namespace configuration options.

`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.

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. The `<http>` element is the parent for all web-related namespace functionality. The `<intercept-url>` element defines a `pattern` which is matched against the URLs of incoming requests using an ant path style syntax[2]. The `access` attribute defines the access requirements for requests matching the given pattern. With the default configuration, this is typically a comma-separated list of roles, one of which a user must have to be allowed to make the request. The prefix "ROLE_" is a marker which indicates that a simple comparison with the user's authorities should be made. In other words, a normal role-based check should be used. Access-control in Spring Security is not limited to the use of simple roles (hence the use of the prefix to differentiate between different types of security attributes). We'll see later how the interpretation can vary[3].

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. You can also add a `method` attribute to limit the match to a particular HTTP method (`GET`, `POST`, `PUT` etc.). If a request matches multiple patterns, the method-specific match will take precedence regardless of ordering.

[2]See the section on Request Matching in the Web Application Infrastructure chapter for more details on how matches are actually performed.

[3]The interpretation of the comma-separated values in the `access` attribute depends on the implementation of the AccessDecisionManager which is used. In Spring Security 3.0, the attribute can also be populated with an EL expression.

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

```
<authentication-manager>
  <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>
</authentication-manager>
```

If you are familiar with pre-namespace 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 problems 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`. All `authentication-provider` elements must be children of the `<authentication-manager>` element, which creates a `ProviderManager` and registers the authentication providers with it. You can find more detailed information on the beans that are created in the namespace appendix. It's worth cross-checking this if you want to start understanding what the important classes in the framework are and how they are used, particularly if you want to customise things later.

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 on the file format. Using the `<authentication-provider>` element means that the user information will be used by the authentication manager to process authentication requests. You can have multiple `<authentication-provider>` elements to define different authentication sources and each will be consulted in turn.

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-based login processing is automatically enabled.

What does `auto-config` Include?

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

```
<http>
  <form-login />
```

```
    <http-basic />
    <logout />
</http>
```

These other elements are responsible for setting up form-login, basic authentication and logout handling services respectively [4] . They each have attributes which can be used to alter their behaviour.

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 after loggin in and so on. However, the namespace offers plenty of support 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*" access="IS_AUTHENTICATED_ANONYMOUSLY"/>
  <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 available to anonymous users [5]. Otherwise the request would be matched by the pattern `/**` and it wouldn't be possible to access the login page itself! This is a common configuration error and will result in an infinite loop in the application. Spring Security will emit a warning in the log if your login page appears to be secured. It is also possible to have all requests matching a particular pattern bypass the security filter chain completely:

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

It's important to realise that these requests will be completely oblivious to any further Spring Security web-related configuration or additional attributes such as `requires-channel`, so you will not be able to access information on the current user or call secured methods during the request. Use `access='IS_AUTHENTICATED_ANONYMOUSLY'` as an alternative if you still want the security filter chain to be applied.

[4] In versions prior to 3.0, this list also included remember-me functionality. This could cause some confusing errors with some configurations and was removed in 3.0. In 3.0, the addition of an `AnonymousAuthenticationFilter` is part of the default `<http>` configuration, so the `<anonymous />` element is added regardless of whether `auto-config` is enabled.

[5] See the chapter on anonymous authentication and also the AuthenticatedVoter class for more details on how the value `IS_AUTHENTICATED_ANONYMOUSLY` is processed.

Note

Using `filters="none"` operates by creating an empty filter chain in Spring Security's `FilterChainProxy`, whereas the access attributes are used to configure the `FilterSecurityInterceptor` in the single filter chain which is created by the namespace configuration. The two are applied independently, so if you have an access contraint for a sub-pattern of a pattern which has a `filters="none"` attribute, the access constraint will be ignored, even if it is listed first. It isn't possible to apply a `filters="none"` attribute to the pattern `/**` since this is used by the namespace filter chain. In version 3.1 things are more flexible. You can define multiple filter chains and the `filters` attribute is no longer supported.

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.

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>
```

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

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

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

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

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

You can also use standard `AuthenticationProvider` beans as follows

```
<authentication-manager>
  <authentication-provider ref='myAuthenticationProvider'/>
</authentication-manager>
```

where `myAuthenticationProvider` is the name of a bean in your application context which implements `AuthenticationProvider`. You can use multiple `authentication-provider` elements, in which case they will be checked in the order they are declared when attempting to authenticated a user. See Section 2.6, “The Authentication Manager and the Namespace” for more on information on how the Spring Security `AuthenticationManager` is configured using the namespace.

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-manager>
  <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>
</authentication-manager>
```

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

Remember-Me Authentication

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

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>
```

Session Management

Detecting Timeouts

You can configure Spring Security to detect the submission of an invalid session ID and redirect the user to an appropriate URL. This is achieved through the `session-management` element:

```
<http>
  ...
  <session-management invalid-session-url="/sessionTimeout.htm" />
</http>
```

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.web.session.HttpSessionEventPublisher
  </listener-class>
</listener>
```

Then add the following lines to your application context:

```
<http>
  ...
  <session-management>
      <concurrency-control max-sessions="1" />
  </session-management>
</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>
  ...
  <session-management>
      <concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />
```

```
    </session-management>
  </http>
```

The second login will then be rejected. By “rejected”, we mean that the user will be sent to the `authentication-failure-url` if form-based login is being used. If the second authentication takes place through another non-interactive mechanism, such as “remember-me”, an “unauthorized” (402) error will be sent to the client. If instead you want to use an error page, you can add the attribute `session-authentication-error-url` to the `session-management` element.

If you are using a customized authentication filter for form-based login, then you have to configure concurrent session control support explicitly. More details can be found in the Session Management chapter.

Session Fixation Attack Protection

Session fixation [http://en.wikipedia.org/wiki/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 `<session-management>`, 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.

OpenID Support

The namespace supports OpenID [http://openid.net/] 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/" authorities="ROLE_USER" />
```

You should be able to login using the `myopenid.com` site to authenticate. It is also possible to select a specific `UserDetailsService` bean for use OpenID by setting the `user-service-ref` attribute on the `openid-login` element. See the previous section on authentication providers for more information. Note that we have omitted the password attribute from the above user configuration, since this set of user data is only being used to load the authorities for the user. A random password

will be generate internally, preventing you from accidentally using this user data as an authentication source elsewhere in your configuration.

Attribute Exchange

Support for OpenID attribute exchange [http://openid.net/specs/openid-attribute-exchange-1_0.html]. As an example, the following configuration would attempt to retrieve the email and full name from the OpenID provider, for use by the application:

```
<openid-login>
  <attribute-exchange>
    <openid-attribute name="email" type="http://axschema.org/contact/email" required="true" />
    <openid-attribute name="name" type="http://axschema.org/namePerson" />
  </attribute-exchange>
</openid-login>
```

The “type” of each OpenID attribute is a URI, determined by a particular schema, in this case http://axschema.org/. If an attribute must be retrieved for successful authentication, the `required` attribute can be set. The exact schema and attributes supported will depend on your OpenID provider. The attribute values are returned as part of the authentication process and can be accessed afterwards using the following code:

```
OpenIDAuthenticationToken token = (OpenIDAuthenticationToken)SecurityContextHolder.getContext().getAuthentic
List<OpenIDAttribute> attributes = token.getAttributes();
```

The `OpenIDAttribute` contains the attribute type and the retrieved value (or values in the case of multi-valued attributes). We'll see more about how the `SecurityContextHolder` class is used when we look at core Spring Security components in the technical overview chapter.

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 `UsernamePasswordAuthenticationFilter` which is created by the `<form-login>` element, taking advantage of some of the extra configuration options which are available by using the bean explicitly. 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. When the application context is being created, the filter beans are sorted by the namespace handling code and the standard Spring Security filters each have an alias in the namespace and a well-known position.

Note

In previous versions, the sorting took place after the filter instances had been created, during post-processing of the application context. In version 3.0+ the sorting is now done at the bean metadata level, before the classes have been instantiated. This has implications for how you add your own filters to the stack as the entire filter list must be known during the parsing of the `<http>` element, so the syntax has changed slightly in 3.0.

The filters, aliases and namespace elements/attributes which create the filters are shown in Table 2.1, “Standard Filter Aliases and Ordering”. The filters are listed in the order in which they occur in the filter chain.

Table 2.1. Standard Filter Aliases and Ordering

Alias	Filter Class	Namespace Element or Attribute
CHANNEL_FILTER	`ChannelProcessingFilter`	`http/intercept-url@requires-channel`
SECURITY_CONTEXT_FILTER	`SecurityContextPersistenceFilter`	`http`
CONCURRENT_SESSION_FILTER	`ConcurrentSessionFilter`	`session-management/concurrency-control`
LOGOUT_FILTER	`LogoutFilter`	`http/logout`
X509_FILTER	`X509AuthenticationFilter`	`http/x509`
PRE_AUTH_FILTER	`AstractPreAuthenticatedProcessingFilter` Subclasses	N/A
CAS_FILTER	`CasAuthenticationFilter`	N/A
FORM_LOGIN_FILTER	`UsernamePasswordAuthenticationFilter`	`http/form-login`
BASIC_AUTH_FILTER	`BasicAuthenticationFilter`	`http/http-basic`
SERVLET_API_SUPPORT_FILTER	`SecurityContextHolderAwareRequestFilter`	`http/@servlet-api-provision`
REMEMBER_ME_FILTER	`RememberMeAuthenticationFilter`	`http/remember-me`
ANONYMOUS_FILTER	`AnonymousAuthenticationFilter`	`http/anonymous`
SESSION_MANAGEMENT_FILTER	`SessionManagementFilter`	`session-management`
EXCEPTION_TRANSLATION_FILTER	`ExceptionTranslationFilter`	`http`
FILTER_SECURITY_INTERCEPTOR	`FilterSecurityInterceptor`	`http`
SWITCH_USER_FILTER	`SwitchUserFilter`	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:

```
<http>
   <custom-filter position="FORM_LOGIN_FILTER" ref="myFilter" />
</http>

<beans:bean id="myFilter" class="com.mycompany.MySpecialAuthenticationFilter"/>
```

You can also use the `after` or `before` attributes 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 - `SecurityContextPersistenceFilter`, `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.

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.

2.4 Method Security

From version 2.0 onwards Spring Security has improved support substantially for adding security to your service layer methods. It provides support for JSR-250 annotation security as well as the framework's original `@Secured` annotation. From 3.0 you can also make use of new expression-based annotations. 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.

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 Spring Security's `@Secured`:

```
<global-method-security secured-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:

```
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);
}
```

Support for JSR-250 annotations can be enabled using

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

These are standards-based and allow simple role-based constraints to be applied but do not have the power Spring Security's native annotations. To use the new expression-based syntax, you would use

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

and the equivalent Java code would be

```
public interface BankService {

  @PreAuthorize("isAnonymous()")
  public Account readAccount(Long id);

  @PreAuthorize("isAnonymous()")
  public Account[] findAccounts();

  @PreAuthorize("hasAuthority('ROLE_TELLER')")
  public Account post(Account account, double amount);
}
```

Expression-based annotations are a good choice if you need to define simple rules that go beyond checking the role names against the user's list of authorities. You can enable more than one type of annotation in the same application, but you should avoid mixing annotations types in the same interface or class to avoid confusion.

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.

2.5 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`. You can find out more about these in the chapter on authorization.

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 Authentication Manager and the Namespace

The main interface which provides authentication services in Spring Security is the `AuthenticationManager`. This is usually an instance of Spring Security's `ProviderManager` class, which you may already be familiar with if you've used the framework before. If not, it will be covered later, in the technical overview chapter. The bean instance is registered using the `authentication-manager` namespace element. You can't use a custom `AuthenticationManager` 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 `AuthenticationProvider`s that are used.

You may want to register additional `AuthenticationProvider` beans with the `ProviderManager` and you can do this using the `<authentication-provider>` element with the `ref` attribute, where the value of the attribute is the name of the provider bean you want to add. For example:

```
<authentication-manager>
  <authentication-provider ref="casAuthenticationProvider"/>
</authentication-manager>

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

Another common requirement is that another bean in the context may require a reference to the `AuthenticationManager`. You can easily register an alias for the `AuthenticationManager` and use this name elsewhere in your application context.

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

<bean id="customizedFormLoginFilter"
      class="com.somecompany.security.web.CustomFormLoginFilter">
   <property name="authenticationManager" ref="authenticationManager"/>
   ...
</bean>
```

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.springsource.org/security/ [http://www.springsource.org/security/] if you need it. All paths referred to in this chapter are relative to the project source directory.

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-3.0.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 an advanced example in that it illustrates the more powerful features of domain object access control lists (ACLs) in addition to basic application security. The application provides an interface with which the users are able to administer a simple database of contacts (the domain objects).

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-3.0.x.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:

```
Security Debug Information

Authentication object is of type:
org.springframework.security.authentication.UsernamePasswordAuthenticationToken

Authentication object as a String:
```

```
org.springframework.security.authentication.UsernamePasswordAuthenticationToken@1f127853:
Principal: org.springframework.security.core.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.web.authentication.WebAuthenticationDetails@0: \
RemoteIpAddress: 127.0.0.1; SessionId: 8fkp8t83ohar; \
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`.

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.springsource.org/browse/SEC.

4.2 Becoming Involved

We welcome your involvement in the 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.springsource.org` to discuss Spring Security with other users of the framework. Remember to use JIRA for bug reports, as explained above.

Part II. Architecture and Implementation

Once you are familiar with setting up and running some namespace-configuration based applications, you may wish to develop more of an understanding of how the framework actually works behind the namespace facade. 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 look at some of these and see how they work together to support authentication and access-control within Spring Security.

5.1 Runtime Environment

Spring Security 3.0 requires a Java 5.0 Runtime Environment or higher. 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. All the required files will be contained within your application.

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 Core Components

In Spring Security 3.0, the contents of the `spring-security-core` jar were stripped down to the bare minimum. It no longer contains any code related to web-application security, LDAP or namespace configuration. We'll take a look here at some of the Java types that you'll find in the core module. They represent the building blocks of the the framework, so if you ever need to go beyond a simple namespace configuration then it's important that you understand what they are, even if you don't actually need to interact with them directly.

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. `SecurityContextHolder` can be configured with a strategy on startup to specify how you would like the context to be stored. For a standalone application you would use the `SecurityContextHolder.MODE_GLOBAL` strategy. 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, the second is to 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.

Obtaining information about the current user

Inside the `SecurityContextHolder` we store details of the principal currently interacting with the application. Spring Security uses an `Authentication` object to represent this information. You won't normally need to create an `Authentication` object yourself, but 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 currently authenticated user, for example:

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

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

The object returned by the call to `getContext()` is an instance of the `SecurityContext` interface. This is the object that is kept in thread-local storage. As we'll see below, most authentication mechanisms withing Spring Security return an instance of `UserDetails` as the principal.

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`:

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

This is the most common approach to loading information for a user within Spring Security and you will see it used throughout the framework whenever information on a user is required.

On successful authentication, `UserDetails` is used to build the `Authentication` object that is stored in the `SecurityContextHolder` (more on this below). The good news is that we provide a number of `UserDetailsService` implementations, including one that uses an in-memory map (`InMemoryDaoImpl`) and another that uses JDBC (`JdbcDaoImpl`). Most users tend to write their own, though, with their implementations often simply sitting on top of an existing Data Access Object (DAO) that represents their employees, customers, or other users of the application. Remember the advantage that whatever your `UserDetailsService` returns can always be obtained from the `SecurityContextHolder` using the above code fragment.

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.

Summary

Just to recap, the major building blocks of Spring Security that we've seen so far are:

- `SecurityContextHolder`, to provide access to the `SecurityContext`.
- `SecurityContext`, to hold the `Authentication` and possibly request-specific security information.
- `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 or other source source of security data.
- `UserDetailsService`, to create a `UserDetails` when passed in a `String`-based username (or certificate ID or the like).

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

Spring Security can participate in many different authentication environments. While 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.

What is authentication in Spring Security?

Let's consider a standard authentication scenario that everyone is familiar with.

1. A user is prompted to log in with a username and password.

2. The system (successfully) verifies that the password is correct for the username.

3. The context information for that user is obtained (their list of roles and so on).

4. A security context is established for the user

5. The user proceeds, potentially to perform some operation which is potentially protected by an access control mechanism which checks the required permissions for the operation against the current security context information.

The first three items constitute the authentication process so we'll take a look at how these take place within Spring Security.

1. The username and password are obtained and combined into an instance of `UsernamePasswordAuthenticationToken` (an instance of the `Authentication` interface, which we saw earlier).

2. The token is passed to an instance of `AuthenticationManager` for validation.

3. The `AuthenticationManager` returns a fully populated `Authentication` instance on successful authentication.

4. The security context is established by calling `SecurityContextHolder.getContext().setAuthentication(...)`, passing in the returned authentication object.

From that point on, the user is considered to be authenticated. Let's look at some code as an example.

```
import org.springframework.security.authentication.*;
import org.springframework.security.core.*;
import org.springframework.security.core.authority.GrantedAuthorityImpl;
import org.springframework.security.core.context.SecurityContextHolder;

public class AuthenticationExample {
  private static AuthenticationManager am = new SampleAuthenticationManager();

  public static void main(String[] args) throws Exception {
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

    while(true) {
      System.out.println("Please enter your username:");
      String name = in.readLine();
      System.out.println("Please enter your password:");
      String password = in.readLine();
      try {
        Authentication request = new UsernamePasswordAuthenticationToken(name, password);
        Authentication result = am.authenticate(request);
        SecurityContextHolder.getContext().setAuthentication(result);
        break;
      } catch(AuthenticationException e) {
        System.out.println("Authentication failed: " + e.getMessage());
      }
    }
    System.out.println("Successfully authenticated. Security context contains: " +
              SecurityContextHolder.getContext().getAuthentication());
  }
}

class SampleAuthenticationManager implements AuthenticationManager {
  static final List<GrantedAuthority> AUTHORITIES = new ArrayList<GrantedAuthority>();
```

```
  static {
    AUTHORITIES.add(new GrantedAuthorityImpl("ROLE_USER"));
  }

  public Authentication authenticate(Authentication auth) throws AuthenticationException {
    if (auth.getName().equals(auth.getCredentials())) {
      return new UsernamePasswordAuthenticationToken(auth.getName(),
        auth.getCredentials(), AUTHORITIES);
      }
      throw new BadCredentialsException("Bad Credentials");
  }
}
```

Here we have written a little program that asks the user to enter a username and password and performs the above sequence. The `AuthenticationManager` which we've implemented here will authenticate any user whose username and password are the same. It assigns a single role to every user. The output from the above will be something like:

```
Please enter your username:
bob
Please enter your password:
password
Authentication failed: Bad Credentials
Please enter your username:
bob
Please enter your password:
bob
Successfully authenticated. Security context contains: \
 org.springframework.security.authentication.UsernamePasswordAuthenticationToken@441d0230: \
 Principal: bob; Password: [PROTECTED]; \
 Authenticated: true; Details: null; \
 Granted Authorities: ROLE_USER
```

Note that you don't normally need to write any code like this. The process will normally occur internally, in a web authentication filter for example. We've just included the code here to show that the question of what actually constitutes authentication in Spring Security has quite a simple answer. A user is authenticated when the `SecurityContextHolder` contains a fully populated `Authentication` object.

Setting the SecurityContextHolder Contents Directly

In fact, Spring Security doesn't mind how you put the `Authentication` object inside the `SecurityContextHolder`. The only critical requirement is that the `SecurityContextHolder` contains an `Authentication` which represents a principal before the `AbstractSecurityInterceptor` (which we'll see more about later) needs to authorize a user operation.

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 situations like this 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 into the `SecurityContextHolder`.

If you're wondering how the `AuthenticationManager` manager is implemented in a real world example, we'll look at that in the core services chapter.

5.4 Authentication in a Web Application

Now let's explore the situation where you are using Spring Security in a web application (without `web.xml` security enabled). How is a user authenticated and the security context established?

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 (via a BASIC authentication dialogue box, a cookie, a X.509 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` and an “authentication mechanism”, which is responsible for calling the `AuthenticationManager` which we saw in the previous section.

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).

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 typically performs one of the actions described in step 3.

Authentication Mechanism

Once your browser submits 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), referring to it as the “authentication mechanism”. Examples are form-base login and Basic authentication. Once the authentication details have been collected from the user agent, an `Authentication` “request” object is built and then presented to the `AuthenticationManager`.

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 `AuthenticationManager` rejected the request, the authentication mechanism will ask the user agent to retry (step two above).

Storing the `SecurityContext` between requests

Depending on the type of application, there may need to be a strategy in place to store the security context between user operations. In a typical web application, a user logs in once and is subsequently identified by their session Id. The server caches the principal information for the duration session. In Spring Security, the responsibility for storing the `SecurityContext` between requests falls to the `SecurityContextPersistenceFilter`, which by default stores the context as an `HttpSession` attribute between HTTP requests. It restores the context to the `SecurityContextHolder` for each request and, crucially, clears the `SecurityContextHolder` when the request completes. You shouldn't interact directly with the `HttpSession` for security purposes. There is simply no justification for doing so - always use the `SecurityContextHolder` instead.

Many other types of application (for example, a stateless RESTful web service) do not use HTTP sessions and will re-authenticate on every request. However, it is still important that

the `SecurityContextPersistenceFilter` is included in the chain to make sure that the `SecurityContextHolder` is cleared after each request.

Note

In an application which receives concurrent requests in a single session, the same `SecurityContext` instance will be shared between threads. Even though a `ThreadLocal` is being used, it is the same instance that is retrieved from the `HttpSession` for each thread. This has implications if you wish to temporarily change the context under which a thread is running. If you just use `SecurityContextHolder.getContext().setAuthentication(anAuthentication)`, then the `Authentication` object will change in *all* concurrent threads which share the same `SecurityContext` instance. You can customize the behaviour of `SecurityContextPersistenceFilter` to create a completely new `SecurityContext` for each request, preventing changes in one thread from affecting another. Alternatively you can create a new instance just at the point where you temporarily change the context. The method `SecurityContextHolder.createEmptyContext()` always returns a new context instance.

5.5 Access-Control (Authorization) in Spring Security

The main interface responsible for making access-control decisions in Spring Security is the `AccessDecisionManager`. It has a `decide` method which takes an `Authentication` object representing the principal requesting access, a "secure object" (see below) and a list of security metadata attributes which apply for the object (such as a list of roles which are required for access to be granted).

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. If you just need to secure method invocations in the services layer, Spring's standard AOP 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 pattern is to perform some web request authorization, coupled with some Spring AOP method invocation authorization on the services layer.

Secure Objects and the `AbstractSecurityInterceptor`

So what *is* a “secure object” anyway? Spring Security uses the term 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.

Each supported secure object type has its own interceptor 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 invocation to proceed (assuming access was granted)
5. Call the `AfterInvocationManager` if configured, once the invocation has returned.

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 are represented by the interface `ConfigAttribute` within the framework. 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 a `SecurityMetadataSource` 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. For example, when we saw something like `<intercept-url pattern='/secure/**' access='ROLE_A,ROLE_B'/>` in the namespace introduction, this is saying that the configuration attributes `ROLE_A` and `ROLE_B` apply to web requests matching the given pattern. In practice, with the default `AccessDecisionManager` configuration, this means that anyone who has a `GrantedAuthority` matching either of these two attributes will be allowed access. Strictly speaking though, they are just attributes and the interpretation is dependent on the `AccessDecisionManager` implementation. The use of the prefix `ROLE_` is a marker to indicate that these attributes are roles and should be consumed by Spring Security's `RoleVoter`. This is only relevant when a voter-based `AccessDecisionManager` is in use. We'll see how the `AccessDecisionManager` is implemented in the authorization chapter.

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.

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 as it chooses.

`AbstractSecurityInterceptor` and its related objects are shown in Figure 5.1, “Security interceptors and the “secure object” model”.

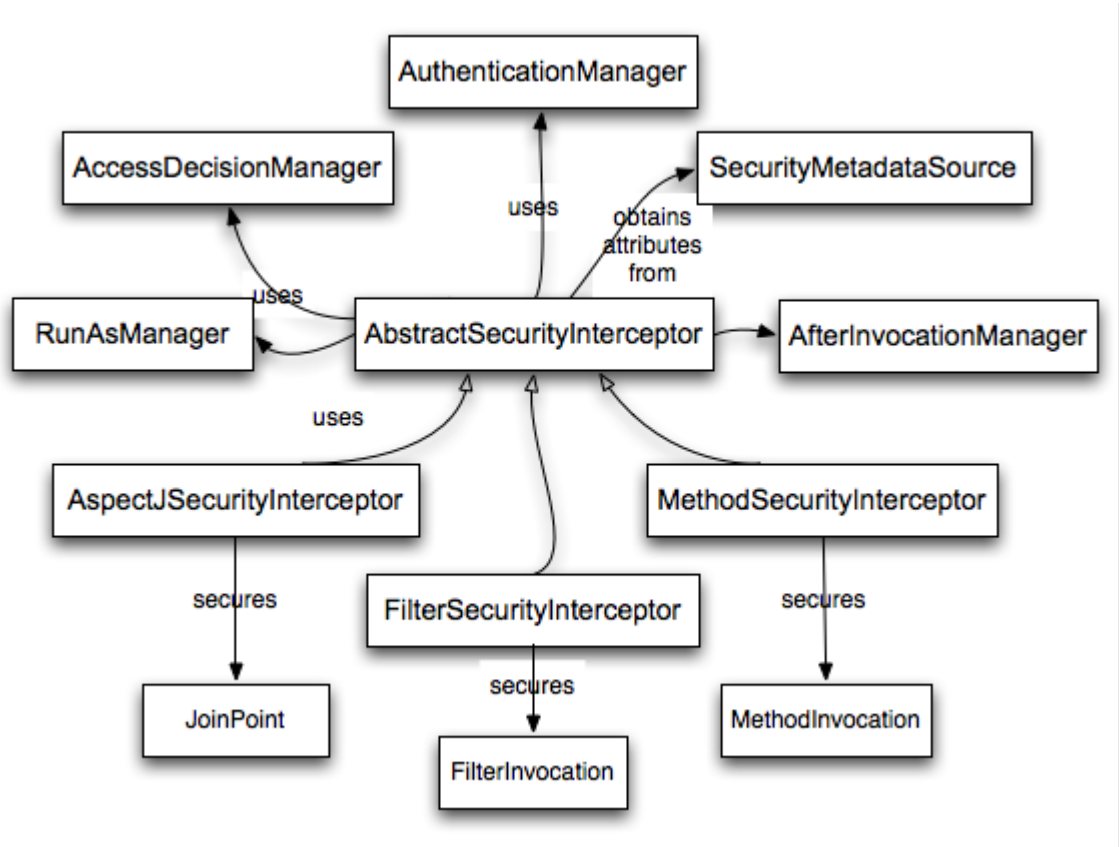

Figure 5.1. Security interceptors and the “secure object” model

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.

5.6 Localization

Spring Security supports localization of exception messages that end users are likely to see. If your application is designed for English-speaking 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 the Spring Framework documentation for further details on using `LocaleContextHolder`.

Now that we have a high-level overview of the Spring Security architecture and its core classes, let's take a closer look at one or two of the core interfaces and their implementations, in particular the `AuthenticationManager`, `UserDetailsService` and the `AccessDecisionManager`. These crop up regularly throughout the remainder of this document so it's important you know how they are configured and how they operate.

6.1 The `AuthenticationManager`, `ProviderManager` and `AuthenticationProviders`

The `AuthenticationManager` is just an interface, so the implementation can be anything we choose, but how does it work in practice? What if we need to check multiple authentication databases or a combination of different authentication services such as a database and an LDAP server?

The default implementation in Spring Security is called `ProviderManager` and rather than handling the authentication request itself, it delegates to a list of configured `AuthenticationProvider`s, each of which is queried in turn to see if it can perform the authentication. Each 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 chapter and refresh your memory. The most common approach to verifying an authentication request is to load the corresponding `UserDetails` and check the loaded password against the one that has been entered by the user. This is the approach used by the `DaoAuthenticationProvider` (see below). The loaded `UserDetails` object - and particularly the `GrantedAuthority`s it contains - will be used when building the fully populated `Authentication` object which is returned from a successful authentication and stored in the `SecurityContext`.

If you are using the namespace, an instance of `ProviderManager` is created and maintained internally, and you add providers to it by using the namespace authentication provider elements (see the namespace chapter). In this case, you should not declare a `ProviderManager` bean in your application context. However, if you are not using the namespace then you would declare it like so:

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

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

Authentication mechanisms such as a web form-login processing filter are injected with a reference to the `ProviderManager` and will call it to handle their authentication requests. The providers you require will sometimes be interchangeable with the authentication mechanisms, while at other times they will depend on a specific authentication mechanism. For example, `DaoAuthenticationProvider` and `LdapAuthenticationProvider` are compatible with any mechanism which submits a simple username/password authentication request and so will work with form-based logins or HTTP Basic authentication. On the other hand, some authentication mechanisms create an authentication request object which can only be interpreted by a single type of `AuthenticationProvider`. An example of this would be JA-SIG CAS, which uses the notion of a service ticket and so can therefore only be authenticated by a `CasAuthenticationProvider`. You needn't be too concerned about this, because if you forget to register a suitable provider, you'll simply receive a `ProviderNotFoundException` when an attempt to authenticate is made.

`DaoAuthenticationProvider`

The simplest `AuthenticationProvider` implemented by Spring Security is `DaoAuthenticationProvider`, which is also one of the earliest supported by the framework. It leverages a `UserDetailsService` (as a DAO) in order to lookup the username, password and `GrantedAuthority`s. It authenticates the user simply by comparing the password submitted in a `UsernamePasswordAuthenticationToken` against the one loaded by the `UserDetailsService`. Configuring the provider is quite simple:

```
<bean id="daoAuthenticationProvider"
    class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="inMemoryDaoImpl"/>
  <property name="saltSource" ref="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. These will be discussed in more detail below.

Erasing Credentials on Successful Authentication

From Spring Security 3.0.3, you can configure the `ProviderManager` will attempt to clear any sensitive credentials information from the `Authentication` object which is returned by a successful authentication request, to prevent information like passwords being retained longer than necessary. This feature is controlled by the `eraseCredentialsAfterAuthentication` property on `ProviderManager`. It is off by default. See the Javadoc for more information.

This may cause issues when you are using a cache of user objects, for example, to improve performance in a stateless application. If the `Authentication` contains a reference to an object in the cache (such as a `UserDetails` instance) and this has its credentials removed, then it will no longer be possible to authenticate against the cached value. You need to take this into account if you are using a cache.

An obvious solution is to make a copy of the object first, either in the cache implementation or in the `AuthenticationProvider` which creates the returned `Authentication` object.

6.2 `UserDetailsService` Implementations

As mentioned in the earlier in this reference guide, most authentication providers take advantage of the `UserDetails` and `UserDetailsService` interfaces. Recall that the contract for `UserDetailsService` is a single method:

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

The returned `UserDetails` is an interface that provides getters that guarantee non-null provision of authentication information such as the username, password, granted authorities and whether the user account 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. They may use the returned `UserDetails` object just for its `GrantedAuthority` information, because some other system (like LDAP or X.509 or CAS etc) has undertaken the responsibility of actually validating the credentials.

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.

In-Memory Authentication

Is easy to use create a custom `UserDetailsService` implementation that extracts information from a persistence engine of choice, but many applications do not require such complexity. This is particularly true if you're building a prototype application 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 supports 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
```

`JdbcDaoImpl`

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.core.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 with any other Spring configuration.

Authority Groups

By default, `JdbcDaoImpl` loads the authorities for a single user with the assumption that the authorities are mapped directly to users (see the database schema appendix). An alternative approach is to partition the authorities into groups and assign groups to the user. Some people prefer this approach as a means of administering user rights. See the `JdbcDaoImpl` Javadoc for more information on how to enable the use of group authorities. The group schema is also included in the appendix.

6.3 Password Encoding

Spring Security's `PasswordEncoder` interface is used to support the use of passwords which are encoded in some way in persistent storage. This will normally mean that the passwords are “hashed” using a digest algorithm such as MD5 or SHA.

What is a hash?

Password hashing is not unique to Spring Security but is a common source of confusion for users who are not familiar with the concept. A hash (or digest) algorithm is a one-way function which produces a piece of fixed-length output data (the hash) from some input data, such as a password. As an example, the MD5 hash of the string “password” (in hexadecimal) is

```
```

```
5f4dcc3b5aa765d61d8327deb882cf99
```

A hash is “one-way” in the sense that it is very difficult (effectively impossible) to obtain the original input given the hash value, or indeed any possible input which would produce that hash value. This property makes hash values very useful for authentication purposes. They can be stored in your user database as an alternative to plaintext passwords and even if the values are compromised they do not immediately reveal a password which can be used to login. Note that this also means you have no way of recovering the password once it is encoded.

Adding Salt to a Hash

One potential problem with the use of password hashes that it is relatively easy to get round the one-way property of the hash if a common word is used for the input. For example, if you search for the hash value `5f4dcc3b5aa765d61d8327deb882cf99` using google, you will quickly find the original word “password”. In a similar way, an attacker can build a dictionary of hashes from a standard word list and use this to lookup the original password. One way to help prevent this is to have a suitably strong password policy to try to prevent common words from being used. Another is to use a “salt” when calculating the hashes. This is an additional string of known data for each user which is combined with the password before calculating the hash. Ideally the data should be as random as possible, but in practice any salt value is usually preferable to none. Spring Security has a `SaltSource` interface which can be used by an authentication provider to generate a salt value for a particular user. Using a salt means that an attacker has to build a separate dictionary of hashes for each salt value, making the attack more complicated (but not impossible).

Hashing and Authentication

When an authentication provider (such as Spring Security's `DaoAuthenticationProvider` needs to check the password in a submitted authentication request against the known value for a user, and the stored password is encoded in some way, then the submitted value must be encoded using exactly the same algorithm. It's up to you to check that these are compatible as Spring Security has no control over the persistent values. If you add password hashing to your authentication configuration in Spring Security, and your database contains plaintext passwords, then there is no way authentication can succeed. Even if you are aware that your database is using MD5 to encode the passwords, for example, and your application is configured to use Spring Security's `Md5PasswordEncoder`, there are still things that can go wrong. The database may have the passwords encoded in Base 64, for example while the enocoder is using hexadecimal strings (the default)[1]. Alternatively your database may be using upper-case while the output from the encoder is lower-case. Make sure you write a test to check the output from your configured password encoder with a known password and salt combination and check that it matches the database value before going further and attempting to authenticate through your application. For more information on the default method for merging salt and password, see the Javadoc for `BasePasswordEncoder`. If you want to generate encoded passwords directly in Java for storage in your user database, then you can use the `encodePassword` method on the `PasswordEncoder`.

[1] You can configure the encoder to use Base 64 instead of hex by setting the `encodeHashAsBase64` property to `true`. Check the Javadoc for `MessageDigestPasswordEncoder` and its parent classes for more information.

Part III. Web Application Security

Most Spring Security users will be using the framework in applications which make user of HTTP and the Servlet API. In this part, we'll take a look at how Spring Security provides authentication and access-control features for the web layer of an application. We'll look behind the facade of the namespace and see which classes and interfaces are actually assembled to provide web-layer security. In some situations it is necessary to use traditional bean configuration to provide full control over the configuration, so we'll also see how to configure these classes directly without the namespace.

Spring Security's web infrastructure is based entirely on standard servlet filters. It doesn't use servlets or any other servlet-based frameworks (such as Spring MVC) internally, so it has no strong links to any particular web technology. It deals in `HttpServletRequest`s and `HttpServletResponse`s and doesn't care whether the requests come from a browser, a web service client, an `HttpInvoker` or an AJAX application.

Spring Security maintains a filter chain internally where each of the filters has a particular responsibility and filters are added or removed from the configuration depending on which services are required. The ordering of the filters is important as there are dependencies between them. If you have been using namespace configuration, then the filters are automatically configured for you and you don't have to define any Spring beans explicitly but here 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.

7.1 `DelegatingFilterProxy`

When using servlet filters, you obviously need to declare them in your `web.xml`, or they will be ignored by the servlet container. In Spring Security, the filter classes are also Spring beans defined in the application context and thus able to take advantage of Spring's rich dependency-injection facilities and lifecycle interfaces. Spring's `DelegatingFilterProxy` provides the link between `web.xml` and the application context.

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 class 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. Read the Javadoc for `DelegatingFilterProxy` for more information

7.2 `FilterChainProxy`

Spring Security's web infrastructure should only be used by delegating to an instance of `FilterChainProxy`. The security filters should not be used by themselves In theory you could declare each Spring Security filter bean that you require in your application context file and add a corresponding `DelegatingFilterProxy` entry to `web.xml` for each filter, making sure that they

are ordered correctly, but this would be cumbersome and would clutter up the `web.xml` file quickly if you have a lot of filters. `FilterChainProxy` lets us add a single entry to `web.xml` and deal entirely with the application context file for managing our web security beans. It is wired using a `DelegatingFilterProxy`, just like in the example above, but with the `filter-name` set to the bean name “filterChainProxy”. The filter chain is then declared in the application context with the same bean name. Here's an example:

```
<bean id="filterChainProxy" class="org.springframework.security.web.FilterChainProxy">
  <sec:filter-chain-map path-type="ant">
     <sec:filter-chain pattern="/webServices/**" filters="
           securityContextPersistenceFilterWithASCFalse,
           basicAuthenticationFilter,
           exceptionTranslationFilter,
           filterSecurityInterceptor" />
     <sec:filter-chain pattern="/**" filters="
           securityContextPersistenceFilterWithASCTrue,
           formLoginFilter,
           exceptionTranslationFilter,
           filterSecurityInterceptor" />
  </sec:filter-chain-map>
</bean>
```

The namespace element `filter-chain-map` is used to set up the security filter chain(s) which are required within the application[1]. It maps a particular URL pattern to a chain of filters built up from the bean names specified in the `filters` element. 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 `SecurityContextPersistenceFilter`s in the filter chain (`ASC` is short for `allowSessionCreation`, a property of `SecurityContextPersistenceFilter`). As web services will never present a `jsessionid` on future requests, creating `HttpSession`s 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 `SecurityContextPersistenceFilter` (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 `Filter`s if such methods are called against `FilterChainProxy` itself. In this case, `FilterChainProxy` guarantees to only initialize and destroy each `Filter` bean once, no matter how many times it is declared in the filter chain(s). You control the overall choice as to whether these methods are called or not via the `targetFilterLifecycle` initialization parameter of `DelegatingFilterProxy`. By default this property is `false` and servlet container lifecycle invocations are not delegated through `DelegatingFilterProxy`.

[1]Note that you'll need to include the security namespace in your application context XML file in order to use this syntax.

When we looked at how to set up web security using namespace configuration, we used a `DelegatingFilterProxy` with the name “springSecurityFilterChain”. You should now be able to see that this is the name of the `FilterChainProxy` which is created by the namespace.

Bypassing the Filter Chain

As with the namespace, you can use the attribute `filters = "none"` as an alternative to supplying a filter bean list. 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. If you want to make use of the contents of the `SecurityContext` contents during a request, then it must have passed through the security filter chain. Otherwise the `SecurityContextHolder` will not have been populated and the contents will be null.

7.3 Filter Ordering

The order that filters are defined in the chain is very important. Irrespective of which filters you are actually using, the order should be as follows:

1. `ChannelProcessingFilter`, because it might need to redirect to a different protocol
2. `SecurityContextPersistenceFilter`, so a `SecurityContext` can be set up 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)
3. `ConcurrentSessionFilter`, because it uses the `SecurityContextHolder` functionality but needs to update the `SessionRegistry` to reflect ongoing requests from the principal
4. Authentication processing mechanisms - `UsernamePasswordAuthenticationFilter`, `CasAuthenticationFilter`, `BasicAuthenticationFilter` 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. `RememberMeAuthenticationFilter`, 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. `AnonymousAuthenticationFilter`, 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 and raise exceptions when access is denied

7.4 Request Matching and `HttpFirewall`

Spring Security has several areas where patterns you have defined are tested against incoming requests in order to decide how the request should be handled. This occurs when the `FilterChainProxy` decides which filter chain a request should be passed through and also when the `FilterSecurityInterceptor` decides which security constraints apply to a request. It's important to understand what the mechanism is and what URL value is used when testing against the patterns that you define.

The Servlet Specification defines several properties for the `HttpServletRequest` which are accessible via getter methods, and which we might want to match against. These are the `contextPath`, `servletPath`, `pathInfo` and `queryString`. Spring Security is only interested in securing paths within the application, so the `contextPath` is ignored. Unfortunately, the servlet spec does not define exactly what the values of `servletPath` and `pathInfo` will contain for a particular request URI. For example, each path segment of a URL may contain parameters, as defined in RFC 2396 [http://www.ietf.org/rfc/rfc2396.txt][2]. The Specification does not clearly state whether these should be included in the `servletPath` and `pathInfo` values and the behaviour varies between different servlet containers. There is a danger that when an application is deployed in a container which does not strip path parameters from these values, an attacker could add them to the requested URL in order to cause a pattern match to succeed or fail unexpectedly.[3]. Other variations in the incoming URL are also possible. For example, it could contain path-traversal sequences (like `/../`) or multiple forward slashes (`//`) which could also cause pattern-matches to fail. Some containers normalize these out before performing the servlet mapping, but others don't. To protect against issues like these, `FilterChainProxy` uses an `HttpFirewall` strategy to check and wrap the request. Un-normalized requests are automatically rejected by default, and path parameters and duplicate slashes are removed for matching purposes.[4]. It is therefore essential that a `FilterChainProxy` is used to manage the security filter chain. Note that the `servletPath` and `pathInfo` values are decoded by the container, so your application should not have any valid paths which contain semi-colons, as these parts will be removed for matching purposes.

As mentioned above, the default strategy is to use Ant-style paths for matching and this is likely to be the best choice for most users. The strategy is implemented in the class `AntPathRequestMatcher` which uses Spring's `AntPathMatcher` to perform a case-insensitive match of the pattern against the concatenated `servletPath` and `pathInfo`, ignoring the `queryString`.

If for some reason, you need a more powerful matching strategy, you can use regular expressions. The strategy implementation is then `RegexRequestMatcher`. See the Javadoc for this class for more information.

---

[2] You have probably seen this when a browser doesn't support cookies and the `jsessionid` parameter is appended to the URL after a semi-colon. However the RFC allows the presence of these parameters in any path segment of the URL

[3] The original values will be returned once the request leaves the `FilterChainProxy`, so will still be available to the application.

[4] So, for example, an original request path `/secure;hack=1/somefile.html;hack=2` will be returned as `/secure/somefile.html`.

In practice we recommend that you use method security at your service layer, to control access to your application, and do not rely entirely on the use of security constraints defined at the web-application level. URLs change and it is difficult to take account of all the possible URLs that an application might support and how requests might be manipulated. You should try and restrict yourself to using a few simple ant paths which are simple to understand. Always try to use a “deny-by-default” approach where you have a catch-all wildcard (`**`) defined last and denying access.

Security defined at the service layer is much more robust and harder to bypass, so you should always take advantage of Spring Security's method security options.

7.5 Use with other Filter-Based Frameworks

If you're using some other framework that is also filter-based, then you need to make sure that the Spring Security filters come first. This enables the `SecurityContextHolder` to be populated in time for use by the other filters. Examples are the use of SiteMesh to decorate your web pages or a web framework like Wicket which uses a filter to handle its requests.

There are some key filters which will always be used in a web application which uses Spring Security, so we'll look at these and their supporting classes and interfaces first. We won't cover every feature, so be sure to look at the Javadoc for them if you want to get the complete picture.

8.1 `FilterSecurityInterceptor`

We've already seen `FilterSecurityInterceptor` briefly when discussing access-control in general, and we've already used it with the namespace where the `<intercept-url>` elements are combined to configure it internally. Now we'll see how to explicitly configure it for use with a `FilterChainProxy`, along with its companion filter `ExceptionTranslationFilter`. A typical configuration example is shown below:

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

`FilterSecurityInterceptor` is responsible for handling the security of HTTP resources. It requires a reference to an `AuthenticationManager` and an `AccessDecisionManager`. It is also supplied with configuration attributes that apply to different HTTP URL requests. Refer back to the original discussion on these in the technical introduction.

The `FilterSecurityInterceptor` can be configured with configuration attributes in two ways. The first, which is shown above, is using the `<filter-security-metadata-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. Commas are used to delimit the different configuration attributes that apply to each HTTP URL. The second option is to write your own `SecurityMetadataSource`, but this is beyond the scope of this document. Irrespective of the approach used, the `SecurityMetadataSource` is responsible for returning a `List<ConfigAttribute>` containing all of the configuration attributes associated with a single secure HTTP URL.

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

The `SecurityMetadataSource` created by the namespace syntax obtains the configuration attributes for a particular `FilterInvocation` by matching the request URL against the configured `pattern` attributes. This behaves in the same way as it does for namespace configuration. The default is to treat all expressions as Apache Ant paths and regular expressions are also supported for more 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. As an example, the previous configuration using regular expressions instead of Ant paths would be written as follows:

```
<bean id="filterInvocationInterceptor"
     class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="runAsManager" ref="runAsManager"/>
  <property name="securityMetadataSource">
    <security:filter-security-metadata-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-security-metadata-source>
  </property>
</bean>
```

Patterns are always evaluated in the order they are defined. Thus it is important that more specific patterns are defined higher in the list than less specific patterns. 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.

8.2 ExceptionTranslationFilter

The `ExceptionTranslationFilter` sits above the `FilterSecurityInterceptor` in the security filter stack. It doesn't do any actual security enforcement itself, but handles exceptions thrown by the security interceptors and provides suitable and HTTP responses.

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

<bean id="authenticationEntryPoint"
     class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">
  <property name="loginFormUrl" value="/login.jsp"/>
</bean>

<bean id="accessDeniedHandler"
     class="org.springframework.security.web.access.AccessDeniedHandlerImpl">
  <property name="errorPage" value="/accessDenied.htm"/>
</bean>
```

AuthenticationEntryPoint

The AuthenticationEntryPoint will be called if the user requests a secure HTTP resource but they are not authenticated. An appropriate AuthenticationException or AccessDeniedException will be thrown by a security interceptor further down the call stack, triggering the commence method on the entry point. This does the job of presenting the appropriate response to the user so that authentication can begin. The one we've used here is LoginUrlAuthenticationEntryPoint, which redirects the request to a different URL (typically a login page). The actual implementation used will depend on the authentication mechanism you want to be used in your application.

AccessDeniedHandler

What happens if a user is already authenticated an they try to access a protected resource? In normal usage, this shouldn't happen because the application workflow should be restricted to operations to which a user has access. For example, an HTML link to an administration page might be hidden from users who do not have an admin role. You can't rely on hiding links for security though, as there's always a possibility that a user will just enter the URL directly in an attempt to bypass the restrictions. Or they might modify a RESTful URL to change some of the argument values. Your application must be protected against these scenarios or it will definitely be insecure. You will typically use simple web layer security to apply constraints to basic URLs and use more specific method-based security on your service layer interfaces to really nail down what is permissible.

If an AccessDeniedException is thrown and a user has already been authenticated, then this means that an operation has been attempted for which they don't have enough permissions. In this case, ExceptionTranslationFilter will invoke a second strategy, the AccessDeniedHandler. By default, an AccessDeniedHandlerImpl is used, which just sends a 403 (Forbidden) response to the client. Alternatively you can configure an instance explicitly (as in the above example) and set an error page URL which it will forwards the request to [1]. This can be a simple “access denied” page, such as a JSP, or it could be a more complex handler such as an MVC controller. And of course, you can implement the interface yourself and use your own implementation.

It's also possible to supply a custom AccessDeniedHandler when you're using the namespace to configure your application. See the namespace appendix for more details.

8.3 SecurityContextPersistenceFilter

We covered the purpose of this all-important filter in the Technical Overview chapter so you might want to re-read that section at this point. Let's first take a look at how you would configure it for use with a FilterChainProxy. A basic configuration only requires the bean itself

```
<bean id="securityContextPersistenceFilter"
class="org.springframework.security.web.context.SecurityContextPersistenceFilter"/>
```

[1]We use a forward so that the SecurityContextHolder still contains details of the principal, which may be useful for displaying 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.

As we saw previously, this filter has two main tasks. It is responsible for storage of the `SecurityContext` contents between HTTP requests and for clearing the `SecurityContextHolder` when a request is completed. Clearing the `ThreadLocal` in which the context is stored is essential, as it might otherwise be possible for a thread to be replaced into the servlet container's thread pool, with the security context for a particular user still attached. This thread might then be used at a later stage, performing operations with the wrong credentials.

`SecurityContextRepository`

From Spring Security 3.0, the job of loading and storing the security context is now delegated to a separate strategy interface:

```
public interface SecurityContextRepository {
  SecurityContext loadContext(HttpRequestResponseHolder requestResponseHolder);
  void saveContext(SecurityContext context, HttpServletRequest request,
         HttpServletResponse response);
}
```

The `HttpRequestResponseHolder` is simply a container for the incoming request and response objects, allowing the implementation to replace these with wrapper classes. The returned contents will be passed to the filter chain.

The default implementation is `HttpSessionSecurityContextRepository`, which stores the security context as an `HttpSession` attribute [2]. The most important configuration parameter for this implementation is the `allowSessionCreation` property, which defaults to `true`, thus allowing the class to create a session if it needs one to store the security context for an authenticated user (it won't create one unless authentication has taken place and the contents of the security context have changed). If you don't want a session to be created, then you can set this property to `false`:

```
<bean id="securityContextPersistenceFilter"
class="org.springframework.security.web.context.SecurityContextPersistenceFilter">
<property name='securityContextRepository'>
<bean class='org.springframework.security.web.context.HttpSessionSecurityContextRepository'>
  <property name='allowSessionCreation' value='false' />
</bean>
</property>
</bean>
```

Alternatively you could provide a null implementation of the `SecurityContextRepository` interface, which will prevent the security context from being stored, even if a session has already been created during the request.

8.4 `UsernamePasswordAuthenticationFilter`

We've now seen the three main filters which are always present in a Spring Security web configuration. These are also the three which are automatically created by the namespace `<http>` element and cannot

[2] In Spring Security 2.0 and earlier, this filter was called `HttpSessionContextIntegrationFilter` and performed all the work of storing the context was performed by the filter itself. If you were familiar with this class, then most of the configuration options which were available can now be found on `HttpSessionSecurityContextRepository`.

be substituted with alternatives. The only thing that's missing now is an actual authentication mechanism, something that will allow a user to authenticate. This filter is the most commonly used authentication filter and the one that is most often customized [3]. It also provides the implementation used by the `<form-login>` element from the namespace. There are three stages required to configure it.

1. Configure a `LoginUrlAuthenticationEntryPoint` with the URL of the login page, just as we did above, and set it on the `ExceptionTranslationFilter`.

2. Implement the login page (using a JSP or MVC controller).

3. Configure an instance of `UsernamePasswordAuthenticationFilter` in the application context

4. Add the filter bean to your filter chain proxy (making sure you pay attention to the order).

The login form simply contains `j_username` and `j_password` input fields, and posts to the URL that is monitored by the filter (by default this is `/j_spring_security_check`). The basic filter configuration looks something like this:

```
<bean id="authenticationFilter" class=
"org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="filterProcessesUrl" value="/j_spring_security_check"/>
</bean>
```

Application Flow on Authentication Success and Failure

The filter calls the configured `AuthenticationManager` to process each authentication request. The destination following a successful authentication or an authentication failure is controlled by the `AuthenticationSuccessHandler` and `AuthenticationFailureHandler` strategy interfaces, respectively. The filter has properties which allow you to set these so you can customize the behaviour completely [4]. Some standard implementations are supplied such as `SimpleUrlAuthenticationSuccessHandler`, `SavedRequestAwareAuthenticationSuccessHandler`, `SimpleUrlAuthenticationFailureHandler` and `ExceptionMappingAuthenticationFailureHandler`. Have a look at the Javadoc for these classes to see how they work.

If authentication is successful, the resulting `Authentication` object will be placed into the `SecurityContextHolder`. The configured `AuthenticationSuccessHandler` will then be called to either redirect or forward the user to the appropriate destination. By default a `SavedRequestAwareAuthenticationSuccessHandler` is used, which means that the user will be redirected to the original destination they requested before they were asked to login.

[3] For historical reasons, prior to Spring Security 3.0, this filter was called `AuthenticationProcessingFilter` and the entry point was called `AuthenticationProcessingFilterEntryPoint`. Since the framework now supports many different forms of authentication, they have both been given more specific names in 3.0.

[4] In versions prior to 3.0, the application flow at this point had evolved to a stage was controlled by a mix of properties on this class and strategy plugins. The decision was made for 3.0 to refactor the code to make these two strategies entirely responsible.

Note

The `ExceptionTranslationFilter` caches the original request a user makes. When the user authenticates, the request handler makes use of this cached request to obtain the original URL and redirect to it. The original request is then rebuilt and used as an alternative.

If authentication fails, the configured `AuthenticationFailureHandler` will be invoked.

Basic and digest authentiation are alternative authentication mechanisms which are popular in web applications. Basic authentication is often used with stateless clients which pass their credentials on each request. It's quite common to use it in combination with form-based authentication where an application is used through both a browser-based user interface and as a web-service. However, basic authentication transmits the password as plain text so it should only really be used over an encrypted transport layer such as HTTPS.

9.1 `BasicAuthenticationFilter`

`BasicAuthenticationFilter` is responsible for 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 browser user agents (such as Firefox and Internet Explorer). The standard governing HTTP Basic Authentication is defined by RFC 1945, Section 11, and `BasicAuthenticationFilter` 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).

Configuration

To implement HTTP Basic Authentication, you need to add a `BasicAuthenticationFilter` to your filter chain. The application context should contain `BasicAuthenticationFilter` and its required collaborator:

```
<bean id="basicAuthenticationFilter"
  class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>
</bean>

<bean id="authenticationEntryPoint"
  class="org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint">
  <property name="realmName" value="Name Of Your Realm"/>
</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 filter in combination with a `BasicAuthenticationEntryPoint`, 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` as usual.

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.

9.2 DigestAuthenticationFilter

DigestAuthenticationFilter 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's DigestAuthenticationFilter 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 more attractive option if you need to use unencrypted HTTP (i.e. 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.

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 DigestAuthenticatonEntryPoint 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 DigestAuthenticationEntryPoint 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 DigestAuthenticationEntryPoint'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's implementation works reliably with FireFox and Internet Explorer, correctly handling nonce timeouts etc.

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 `DigestAuthenticationFilter` in the filter chain. The application context will need to define the `DigestAuthenticationFilter` and its required collaborators:

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

<bean id="digestEntryPoint" class=
    "org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint">
  <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 `DigestAuthenticationFilter` 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 `DigestAuthenticationEntryPoint`, so that `DigestAuthenticationFilter` 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.

10.1 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.

10.2 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 + ":" + key))

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>
```

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.

10.3 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 va
```

10.4 Remember-Me Interfaces and Implementations

Remember-me authentication is not used with basic authentication, given it is often not used with `HttpSession`s. Remember-me is used with `UsernamePasswordAuthenticationFilter`, and is implemented via hooks in the `AbstractAuthenticationProcessingFilter` 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 successfulAuthentication);
```

Please refer to the JavaDocs for a fuller discussion on what the methods do, although note at this stage that `AbstractAuthenticationProcessingFilter` only calls the `loginFail()` and `loginSuccess()` methods. The `autoLogin()` method is called by `RememberMeAuthenticationFilter` 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 these in turn.

[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.

TokenBasedRememberMeServices

This implementation supports the simpler approach described in Section 10.2, “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 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="rememberMeFilter" class=
 "org.springframework.security.web.authentication.rememberme.RememberMeAuthenticationFilter">
  <property name="rememberMeServices" ref="rememberMeServices"/>
  <property name="authenticationManager" ref="theAuthenticationManager" />
</bean>

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

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

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

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 10.3, “Persistent Token Approach”.

HTTP session related functonality is handled by a combination of the `SessionManagementFilter` and the `SessionAuthenticationStrategy` interface, which the filter delegates to. Typical usage includes session-fixation protection attack prevention, detection of session timeouts and restrictions on how many sessions an authenticated user may have open concurrently.

11.1 SessionManagementFilter

The `SessionManagementFilter` checks the contents of the `SecurityContextRepository` against the current contents of the `SecurityContextHolder` to determine whether a user has been authenticated during the current request, typically by a non-interactive authentication mechanism, such as pre-authentication or remember-me [1]. If the repository contains a security context, the filter does nothing. If it doesn't, and the thread-local `SecurityContext` contains a (non-anonymous) `Authentication` object, the filter assumes they have been authenticated by a previous filter in the stack. It will then invoke the configured `SessionAuthenticationStrategy`.

If the user is not currently authenticated, the filter will check whether an invalid session ID has been requested (because of a timeout, for example) and will redirect to the configured `invalidSessionUrl` if set. The easiest way to configure this is through the namespace, as described earlier.

11.2 `SessionAuthenticationStrategy`

`SessionAuthenticationStrategy` is used by both `SessionManagementFilter` and `AbstractAuthenticationProcessingFilter`, so if you are using a customized form-login class, for example, you will need to inject it into both of these. In this case, a typical configuration, combining the namespace and custom beans might look like this:

```
<http>
  <custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />
  <session-management session-authentication-strategy-ref="sas"/>
</http>

<beans:bean id="myAuthFilter"
    class="org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
  <beans:property name="sessionAuthenticationStrategy" ref="sas" />
  ...
</beans:bean>

<beans:bean id="sas"
    class="org.springframework.security.web.authentication.session.SessionFixationProtectionStrategy"/>
```

[1] Authentication by mechanisms which perform a redirect after authenticating (such as form-login) will not be detected by `SessionManagementFilter`, as the filter will not be invoked during the authenticating request. Session-management functionality has to be handled separately in these cases.

11.3 Concurrency Control

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. You can either expire their previous login or you can report an error when they try to log in again, preventing the second login. Note that if you are using the second approach, a user who has not explicitly logged out (but who has just closed their browser, for example) will not be able to log in again until their original session expires.

Concurrency control is supported by the namespace, so please check the earlier namespace chapter for the simplest configuration. Sometimes you need to customize things though.

The implementation uses a specialized version of `SessionAuthenticationStrategy`, called `ConcurrentSessionControlStrategy`.

Note

Previously the concurrent authentication check was made by the `ProviderManager`, which could be injected with a `ConcurrentSessionController`. The latter would check if the user was attempting to exceed the number of permitted sessions. However, this approach required that an HTTP session be created in advance, which is undesirable. In Spring Security 3, the user is first authenticated by the `AuthenticationManager` and once they are successfully authenticated, a session is created and the check is made whether they are allowed to have another session open.

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

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

In addition, you will need to add the `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. A configuration using the namespace to create the `FilterChainProxy` and other default beans might look like this:

```
<http>
  <custom-filter position="CONCURRENT_SESSION_FILTER" ref="concurrencyFilter" />
  <custom-filter position="FORM_LOGIN_FILTER" ref="myAuthFilter" />

  <session-management session-authentication-strategy-ref="sas"/>
</http>
```

```
<beans:bean id="concurrencyFilter"
    class="org.springframework.security.web.session.ConcurrentSessionFilter">
  <beans:property name="sessionRegistry" ref="sessionRegistry" />
  <beans:property name="expiredUrl" value="/session-expired.htm" />
</beans:bean>

<beans:bean id="myAuthFilter"
    class="org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">
  <beans:property name="sessionAuthenticationStrategy" ref="sas" />
  <beans:property name="authenticationManager" ref="authenticationManager" />
</beans:bean>

<beans:bean id="sas"
    class="org.springframework.security.web.authentication.session.ConcurrentSessionControlStrategy">
  <beans:constructor-arg name="sessionRegistry" ref="sessionRegistry" />
  <beans:property name="maximumSessions" value="1" />
</beans:bean>

<beans:bean id="sessionRegistry" class="org.springframework.security.core.session.SessionRegistryImpl" />
```

Adding the listener to `web.xml` 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. Without it, a user will never be able to log back in again once they have exceeded their session allowance, even if they log out of another session or it times out.

12.1 Overview

It's generally considered good security practice to adopt a “deny-by-default” where you explicitly specify what is allowed and disallow everything else. Defining what is accessible to unauthenticated users is a similar situation, particularly for web applications. Many sites require that users must be authenticated for anything other than a few URLs (for example the home and login pages). In this case it is easiest to define access configuration attributes for these specific URLs rather than have for every secured resource. 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. You could also omit these pages from the filter chain entirely, thus bypassing the access control checks, but this may be undesirable for other reasons, particularly if the pages behave differently for authenticated users.

This is what we mean by anonymous authentication. Note that there is no real conceptual difference between a user who is “anonymously authenticated” and an unauthenticated user. Spring Security's anonymous authentication just gives you a more convenient way to configure your access-control attributes. Calls to servlet API calls such as `getCallerPrincipal`, for example, will still return null even though there is actually an anonymous authentication object in the `SecurityContextHolder`.

There are other situations where anonymous authentication is useful, such as when an auditing interceptor queries the `SecurityContextHolder` to identify which principal was responsible for a given operation. Classes can be authored more robustly if they know the `SecurityContextHolder` always contains an `Authentication` object, and never `null`.

12.2 Configuration

Anonymous authentication support is provided automatically when using the HTTP configuration Spring Security 3.0 and can be customized (or disabled) using the `<anonymous>` element. You don't need to configure the beans described here unless you are using traditional bean configuration.

Three classes that together provide the 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 `AnonymousAuthenticationToken`s are accepted. Finally, there is an `AnonymousAuthenticationFilter`, which is chained after the normal authentication mechanisms and automatically adds 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="anonymousAuthFilter"
    class="org.springframework.security.web.authentication.AnonymousAuthenticationFilter">
  <property name="key" value="foobar"/>
  <property name="userAttribute" value="anonymousUser,ROLE_ANONYMOUS"/>
</bean>
```

```
<bean id="anonymousAuthenticationProvider"
    class="org.springframework.security.authentication.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[1]. 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="filterSecurityInterceptor"
    class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/>
  <property name="securityMetadata">
    <security:filter-security-metadata-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-security-metadata-source>" +
  </property>
</bean>
```

12.3 `AuthenticationTrustResolver`

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 `AccessDeniedException`s. 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.

[1]The use of the `key` property should not be regarded as providing any real security here. It is merely a book-keeping exercise. If you are sharing a `ProviderManager` which contains an `AnonymousAuthenticationProvider` in a scenario where it is possible for an authenticating client to construct the `Authentication` object (such as with RMI invocations), then a malicious client could submit an `AnonymousAuthenticationToken` which it had created itself (with chosen username and authority list). If the `key` is guessable or can be found out, then the token would be accepted by the anonymous provider. This isn't a problem with normal usage but if you are using RMI you would be best to use a customized `ProviderManager` which omits the anonymous provider rather than sharing the one you use for your HTTP authentication mechanisms.

You will often see the `ROLE_ANONYMOUS` attribute in the above interceptor configuration replaced with `IS_AUTHENTICATED_ANONYMOUSLY`, which is effectively the same thing when defining access controls. This is an example of the use of the `AuthenticatedVoter` which we will see in the authorization chapter. It uses an `AuthenticationTrustResolver` to process this particular configuration attribute and grant access to anonymous users. The `AuthenticatedVoter` approach is more powerful, since it allows you to differentiate between anonymous, remember-me and fully-authenticated users. If you don't need this functionality though, then you can stick with `ROLE_ANONYMOUS`, which will be processed by Spring Security's standard `RoleVoter`.

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.

13.1 Authorities

As we saw in the technical overview, all `Authentication` implementations store a list 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 `AccessDecisionManager`s when making authorization decisions.

`GrantedAuthority` is an interface with only one method:

```
String getAuthority();
```

This method allows `AccessDecisionManager`s to obtain a precise `String` representation of the `GrantedAuthority`. By returning a representation as a `String`, a `GrantedAuthority` can be easily "read" by most `AccessDecisionManager`s. 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 difficult, 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 `AuthenticationProvider`s included with the security architecture use `GrantedAuthorityImpl` to populate the `Authentication` object.

13.2 Pre-Invocation Handling

As we've also seen 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`.

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,
    List<ConfigAttribute> config) throws AccessDeniedException;
boolean supports(ConfigAttribute attribute);
```

```
boolean supports(Class clazz);
```

The `AccessDecisionManager`'s `decide` method is passed all the relevant information it needs in order to make 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.

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 13.1, “Voting Decision Manager” illustrates the relevant classes.

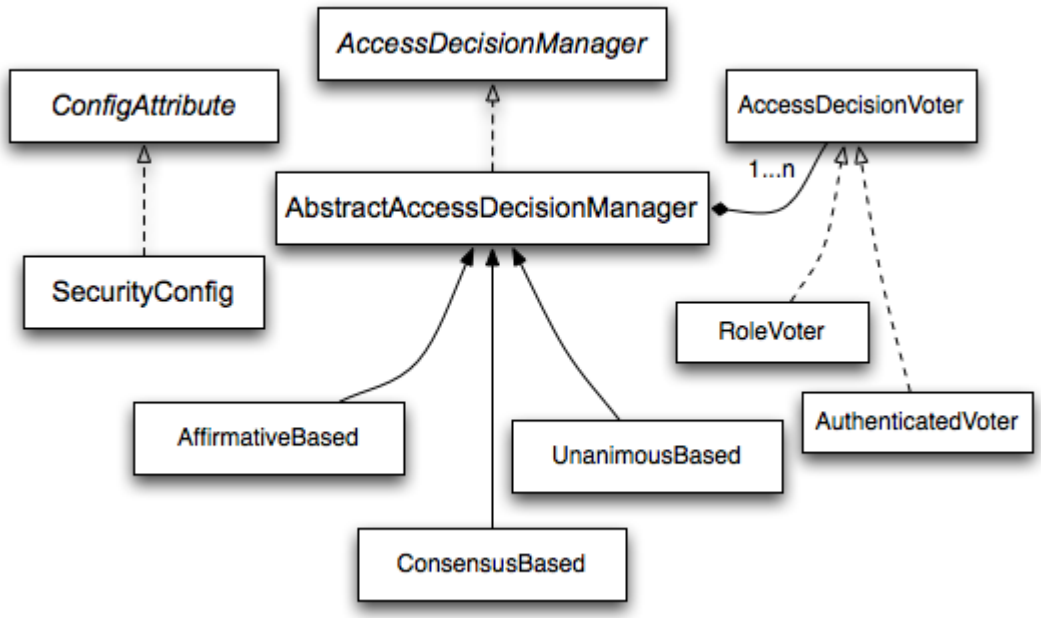

Figure 13.1. Voting Decision Manager

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, List<ConfigAttribute> 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 `AccessDecisionManager`s 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.

`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 the prefix `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.

`AuthenticatedVoter`

Another voter which we've implicitly seen is the `AuthenticatedVoter`, which can be used to differentiate between anonymous, fully-authenticated and remember-me authenticated users. Many sites allow certain limited access under remember-me authentication, but require a user to confirm their identity by logging in for full access.

When we've used the attribute `IS_AUTHENTICATED_ANONYMOUSLY` to grant anonymous access, this attribute was being processed by the `AuthenticatedVoter`. See the Javadoc for this class for more information.

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.

13.3 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 13.2, “After Invocation Implementation” illustrates Spring Security's `AfterInvocationManager` and its concrete implementations.

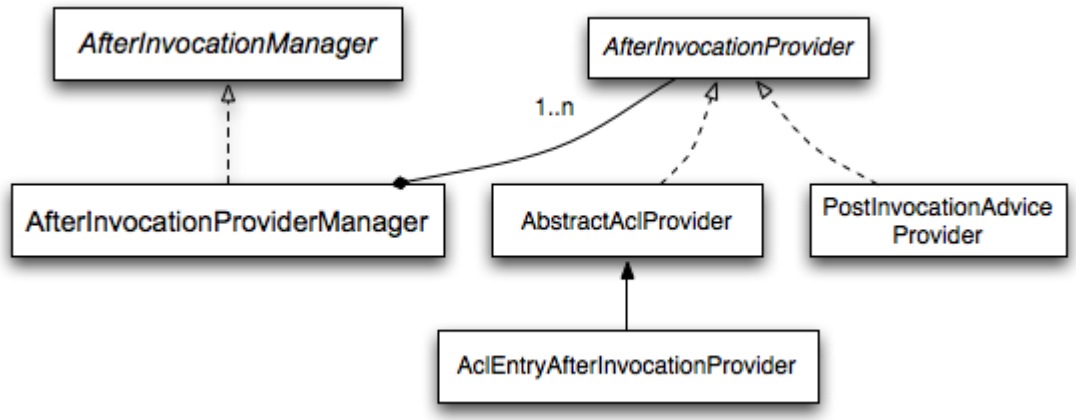

Figure 13.2. After Invocation Implementation

Like many other parts of Spring Security, `AfterInvocationManager` has a single concrete implementation, `AfterInvocationProviderManager`, which polls a list of `AfterInvocationProvider`s. 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.

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.

14.1 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 `MethodSecurityMetadataSource` instance to obtain the configuration attributes that apply to a particular method invocation. `MapBasedMethodSecurityMetadataSource` 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.

Explicit MethodSecurityInterceptor 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.access.intercept.aopalliance.MethodSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="afterInvocationManager" ref="afterInvocationManager"/>
  <property name="securityMetadataSource">
    <value>
      com.mycompany.BankManager.delete*=ROLE_SUPERVISOR
      com.mycompany.BankManager.getBalance=ROLE_TELLER,ROLE_SUPERVISOR
    </value>
  </property>
</bean>
```

14.2 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.access.intercept.aspectj.AspectJSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="accessDecisionManager"/>
  <property name="afterInvocationManager" ref="afterInvocationManager"/>
  <property name="securityMetadataSource">
    <value>
        com.mycompany.BankManager.delete*=ROLE_SUPERVISOR
        com.mycompany.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 `securityMetadataSource`, as the `SecurityMetadataSource` works with `java.lang.reflect.Method`s 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.access.intercept.aspectj.AspectJSecurityInterceptor;
import org.springframework.security.access.intercept.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="security.samples.aspectj.DomainObjectInstanceSecurityAspect"
     factory-method="aspectOf">
  <property name="securityInterceptor" ref="bankManagerSecurity"/>
</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.

Spring Security 3.0 introduced the ability to use Spring EL expressions as an authorization mechanism in addition to the simple use of configuration attributes and access-decision voters which have seen before. Expression-based access control is built on the same architecture but allows complicated boolean logic to be encapsulated in a single expression.

15.1 Overview

Spring Security uses Spring EL for expression support and you should look at how that works if you are interested in understanding the topic in more depth. Expressions are evaluated with a “root object” as part of the evaluation context. Spring Security uses specific classes for web and method security as the root object, in order to provide built-in expressions and access to values such as the current principal.

Common Built-In Expressions

The base class for expression root objects is `SecurityExpressionRoot`. This provides some common expressions which are available in both web and method security.

Table 15.1. Common built-in expressions

Expression	Description
`hasRole([role])`	Returns `true` if the current principal has the specified role.
`hasAnyRole([role1,role2])`	Returns `true` if the current principal has any of the supplied roles (given as a comma-separated list of strings)
`principal`	Allows direct access to the principal object representing the current user
`authentication`	Allows direct access to the current `Authentication` object obtained from the `SecurityContext`
`permitAll`	Always evaluates to `true`
`denyAll`	Always evaluates to `false`
`isAnonymous()`	Returns `true` if the current principal is an anonymous user
`isRememberMe()`	Returns `true` if the current principal is a remember-me user
`isAuthenticated()`	Returns `true` if the user is not anonymous
`isFullyAuthenticated()`	Returns `true` if the user is not an anonymous or a remember-me user

15.2 Web Security Expressions

To use expressions to secure individual URLs, you would first need to set the `use-expressions` attribute in the `<http>` element to `true`. Spring Security will then expect the `access` attributes of

the <intercept-url> elements to contain Spring EL expressions. The expressions should evaluate to a boolean, defining whether access should be allowed or not. For example:

```
<http use-expressions="true">
  <intercept-url pattern="/admin*"
      access="hasRole('admin') and hasIpAddress('192.168.1.0/24')"/>
  ...
</http>
```

Here we have defined that the “admin” area of an application (defined by the URL pattern) should only be available to users who have the granted authority “admin” and whose IP address matches a local subnet. We've already seen the built-in `hasRole` expression in the previous section. The expression `hasIpAddress` is an additional built-in expression which is specific to web security. It is defined by the `WebSecurityExpressionRoot` class, an instance of which is used as the expression root object when evaluation web-access expressions. This object also directly exposed the `HttpServletRequest` object under the name `request` so you can invoke the request directly in an expression.

If expressions are being used, a `WebExpressionVoter` will be added to the `AccessDecisionManager` which is used by the namespace. So if you aren't using the namespace and want to use expressions, you will have to add one of these to your configuration.

15.3 Method Security Expressions

Method security is a bit more complicated than a simple allow or deny rule. Spring Security 3.0 introduced some new annotations in order to allow comprehensive support for the use of expressions.

@Pre and @Post Annotations

There are four annotations which support expression attributes to allow pre and post-invocation authorization checks and also to support filtering of submitted collection arguments or return values. They are `@PreAuthorize`, `@PreFilter`, `@PostAuthorize` and `@PostFilter`. Their use is enabled through the `global-method-security` namespace element:

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

Access Control using @PreAuthorize and @PostAuthorize

The most obviously useful annotation is `@PreAuthorize` which decides whether a method can actually be invoked or not. For example (from the “Contacts” sample application)

```
@PreAuthorize("hasRole('ROLE_USER')")
public void create(Contact contact);
```

which means that access will only be allowed for users with the role "ROLE_USER". Obviously the same thing could easily be achieved using a traditional configuration and a simple configuration attribute for the required role. But what about:

```
@PreAuthorize("hasPermission(#contact, 'admin')")
public void deletePermission(Contact contact, Sid recipient, Permission permission);
```

Here we're actually using a method argument as part of the expression to decide whether the current user has the “admin”permission for the given contact. The built-in `hasPermission()` expression is linked into the Spring Security ACL module through the application context, as we'll see below. You can access any of the method arguments by name as expression variables, provided your code has debug information compiled in. Any Spring-EL functionality is available within the expression, so you can also access properties on the arguments. For example, if you wanted a particular method to only allow access to a user whose username matched that of the contact, you could write

```
@PreAuthorize("#contact.name == authentication.name")
public void doSomething(Contact contact);
```

Here we are accessing another built–in expression, `authentication`, which is the `Authentication` stored in the security context. You can also access its “principal” property directly, using the expression `principal`. The value will often be a `UserDetails` instance, so you might use an expression like `principal.username` or `principal.enabled`.

Less commonly, you may wish to perform an access-control check after the method has been invoked. This can be achieved using the `@PostAuthorize` annotation. To access the return value from a method, use the built–in name `returnObject` in the expression.

Filtering using `@PreFilter` and `@PostFilter`

As you may already be aware, Spring Security supports filtering of collections and arrays and this can now be achieved using expressions. This is most commonly performed on the return value of a method. For example:

```
@PreAuthorize("hasRole('ROLE_USER')")
@PostFilter("hasPermission(filterObject, 'read') or hasPermission(filterObject, 'admin')")
public List<Contact> getAll();
```

When using the `@PostFilter` annotation, Spring Security iterates through the returned collection and removes any elements for which the supplied expression is false. The name `filterObject` refers to the current object in the collection. You can also filter before the method call, using `@PreFilter`, though this is a less common requirement. The syntax is just the same, but if there is more than one argument which is a collection type then you have to select one by name using the `filterTarget` property of this annotation.

Note that filtering is obviously not a substitute for tuning your data retrieval queries. If you are filtering large collections and removing many of the entries then this is likely to be inefficient.

Built-In Expressions

There are some built-in expressions which are specific to method security, which we have already seen in use above. The `filterTarget` and `returnValue` values are simple enough, but the use of the `hasPermission()` expression warrants a closer look.

The `PermissionEvaluator` interface

`hasPermission()` expressions are delegated to an instance of `PermissionEvaluator`. It is intended to bridge between the expression system and Spring Security's ACL system, allowing you to

specify authorization constraints on domain objects, based on abstract permissions. It has no explicit dependencies on the ACL module, so you could swap that out for an alternative implementation if required. The interface has two methods:

```
boolean hasPermission(Authentication authentication, Object targetDomainObject, Object permission);

boolean hasPermission(Authentication authentication, Serializable targetId, String targetType, Object perm
```

which map directly to the available versions of the expression, with the exception that the first argument (the `Authentication` object) is not supplied. The first is used in situations where the domain object, to which access is being controlled, is already loaded. Then expression will return true if the current user has the given permission for that object. The second version is used in cases where the object is not loaded, but its identifier is known. An abstract “type” specifier for the domain object is also required, allowing the correct ACL permissions to be loaded. This has traditionally been the Java class of the object, but does not have to be as long as it is consistent with how the permissions are loaded.

To use `hasPermission()` expressions, you have to explicitly configure a `PermissionEvaluator` in your application context. This would look something like this:

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

<bean id="expressionHandler"
    class="org.springframework.security.access.expression.method.DefaultMethodSecurityExpressionHandler">
      <property name="permissionEvaluator" ref="myPermissionEvaluator"/>
</bean>
```

Where `myPermissionEvaluator` is the bean which implements `PermissionEvaluator`. Usually this will be the implementation from the ACL module which is called `AclPermissionEvaluator`. See the “Contacts” sample application configuration for more details.

Part V. Additional Topics

In this part we cover features which require a knowledge of previous chapters as well as some of the more advanced and less-commonly used features of the framework.

16.1 Overview

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 `Customer`s. If a user might be able to access 5,000 `Customer`s (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.

16.2 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 principal 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.

16.3 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.

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`.

17.1 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 configuration or are planning on writing your own implementation then an understanding of how the provided implementations work will be useful. You will find classes under the org.springframework.security.web.authentication.preauth. We just provide an outline here so you should consult the Javadoc and source where appropriate.

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.

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.

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.

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.

Http403ForbiddenEntryPoint

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.

17.2 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.

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 Siteminder, which passes the username in a header called `SM_USER`. This mechanism is supported by the class `RequestHeaderAuthenticationFilter` 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 username they wished.

Siteminder Example Configuration

A typical configuration using this filter would look like this:

```
  <security:http>
    <!-- Additional http configuration omitted -->
    <security:custom-filter position="PRE_AUTH_FILTER" ref="siteminderFilter" />
  </security:http>

    <bean id="siteminderFilter" class=
"org.springframework.security.web.authentication.preauth.RequestHeaderAuthenticationFilter">
    <property name="principalRequestHeader" value="SM_USER"/>
    <property name="authenticationManager" ref="authenticationManager" />
  </bean>

  <bean id="preauthAuthProvider"
class="org.springframework.security.web.authentication.preauth.PreAuthenticatedAuthenticationProvider">
    <property name="preAuthenticatedUserDetailsService">
      <bean id="userDetailsServiceWrapper"
          class="org.springframework.security.core.userdetails.UserDetailsByNameServiceWrapper">
        <property name="userDetailsService" ref="userDetailsService"/>
      </bean>
    </property>
    </bean>

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

We've assumed here that the security namespace is being used for configuration. It's also assumed that you have added a `UserDetailsService` (called “userDetailsService”) to your configuration to load the user's roles.

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 the section called "J2eeBasedPreAuthenticatedWebAuthenticationDetailsSource".

There is a sample application in the codebase which uses this approach, so get hold of the code from subversion and have a look at the application context file if you are interested. The code is in the `samples/preauth` directory.

18.1 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.

18.2 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).

18.3 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" />
```

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.

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.

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.

18.4 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, `LdapAuthenticationProvider`, 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.

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.

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.

BindAuthenticator

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

PasswordComparisonAuthenticator

The class `PasswordComparisonAuthenticator` implements the password comparison authentication strategy.

Active Directory Authentication

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

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`.

LDAP Search Objects

Often 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`.

`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 [http://java.sun.com/j2se/1.4.2/docs/api/javax/naming/directory/DirContext.html#search(javax.naming.Name,%20java.lang.String,%20java.lang.Object[],%20javax.naming.directory.SearchControls)]. 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.

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.

If you want to use LDAP only for authentication, but load the authorities from a difference source (such as a database) then you can provide your own implementation of this interface and inject that instead.

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.ldap.authentication.LdapAuthenticationProvider">
 <constructor-arg>
   <bean class="org.springframework.security.ldap.authentication.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.userdetails.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.

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,
          Collection<GrantedAuthority> authorities);

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

Only the first method is relevant for authentication. If you provide an implementation of this interface and inject it into the `LdapAuthenticationProvider`, you have control over 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 during authentication. The `username` parameter is the name used to authenticate and the final parameter is the collection of authorities loaded for the user by the configured `LdapAuthoritiesPopulator`.

The way the context data is loaded varies slightly depending on the type of authentication you are using. With the `BindAuthenticator`, 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).

Spring Security has its own taglib which provides basic support for accessing security information and applying security constraints in JSPs.

19.1 Declaring the Taglib

To use any of the tags, you must have the security taglib declared in your JSP:

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

19.2 The `authorize` Tag

This tag is used to determine whether its contents should be evaluated or not. In Spring Security 3.0, it can be used in two ways [1]. The first approach uses a web-security expression, specified in the `access` attribute of the tag. The expression evaluation will be delegated to the `WebSecurityExpressionHandler` defined in the application context (you should have web expressions enabled in your `<http>` namespace configuration to make sure this service is available). So, for example, you might have

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

This content will only be visible to users who have
the "supervisor" authority in their list of <tt>GrantedAuthority</tt>s.

</sec:authorize>
```

A common requirement is to only show a particular link, if the user is actually allowed to click it. How can we determine in advance whether something will be allowed? This tag can also operate in an alternative mode which allows you to define a particular URL as an attribute. If the user is allowed to invoke that URL, then the tag body will be evaluated, otherwise it will be skipped. So you might have something like

```
<sec:authorize url="/admin">

This content will only be visible to users who are authorized to send requests to the "/admin" URL.

</sec:authorize>
```

To use this tag there must also be an instance of `WebInvocationPrivilegeEvaluator` in your application context. If you are using the namespace, one will automatically be registered. This is an instance of `DefaultWebInvocationPrivilegeEvaluator`, which creates a dummy web request for the supplied URL and invokes the security interceptor to see whether the request would succeed or fail. This allows you to delegate to the access-control setup you defined using `intercept-url` declarations within the `<http>` namespace configuration and saves having to duplicate the information (such as the required roles) within your JSPs. This approach can also be combined with a `method` attribute, supplying the HTTP method, for a more specific match.

[1] The legacy options from Spring Security 2.0 are also supported, but discouraged.

19.3 The authenticationTag

This tag allows access to the current Authentication object stored in the security context. It renders a property of the object directly in the JSP. So, for example, if the principal property of the Authentication is an instance of Spring Security's UserDetails object, then using <sec:authentication property="principal.username" /> will render the name of the current user.

Of course, it isn't necessary to use JSP tags for this kind of thing and some people prefer to keep as little logic as possible in the view. You can access the Authentication object in your MVC controller (by calling SecurityContextHolder.getContext().getAuthentication()) and add the data directly to your model for rendering by the view.

19.4 The accesscontrollist Tag

This tag is only valid when used with Spring Security's ACL module. It checks a comma-separated list of required permissions for a specified domain object. If the current user has any of those permissions, then the tag body will be evaluated. If they don't, it will be skipped. An example might be

```
<sec:accesscontrollist hasPermission="1,2" domainObject="someObject">

This will be shown if the user has either of the permissions
represented by the values "1" or "2" on the given object.

</sec:accesscontrollist>
```

The permissions are passed to the PermissionFactory defined in the application context, converting them to ACL Permission instances, so they may be any format which is supported by the factory - they don't have to be integers, they could be strings like READ or WRITE. If no PermissionFactory is found, an instance of DefaultPermissionFactory will be used. The AclServicefrom the application context will be used to load the Acl instance for the supplied object. The Acl will be invoked with the required permissions to check if any of them are granted.

20.1 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.

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

The `CallbackHandler`s and `AuthorityGranter`s are discussed below.

JAAS CallbackHandler

Most JAAS `LoginModule`s 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 `JaasAuthenticationProvider` wraps these `JaasAuthenticationCallbackHandler`s 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 `InternalCallbackHandler`s. If the `LoginModule` requests a callback against the `InternalCallbackHandler`s, the callback is in-turn passed to the `JaasAuthenticationCallbackHandler`s being wrapped.

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 set of `String`s, representing the authorities assigned to the principal. For each returned authority string, the `JaasAuthenticationProvider` creates a `JaasGrantedAuthority` (which implements Spring Security's `GrantedAuthority` interface) containing the authority 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 `AuthorityGranter`s 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.

21.1 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/cas`. You will also need to visit this site to download the CAS Server files.

21.2 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 3.0 supports CAS 3. At the time of writing, the CAS server was at version 3.3.

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 3.3 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 `AuthenticationHandler`s 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.

21.3 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 CAS service:

```
<bean id="serviceProperties"
```

```
        class="org.springframework.security.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 `CasAuthenticationFilter`. 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 (assuming you're using a namespace configuration):

```
<security:http entry-point-ref="casEntryPoint">
   ...
   <security:custom-filter position="CAS_FILTER" ref="casFilter" />
</security:http>

<bean id="casFilter"
      class="org.springframework.security.cas.web.CasAuthenticationFilter">
  <property name="authenticationManager" ref="authenticationManager"/>
</bean>

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

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

The `CasAuthenticationFilter` has very similar properties to the `UsernamePasswordAuthenticationFilter` (used for form-based logins).

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

The `CasAuthenticationEntryPoint` 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:

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

  <bean id="casAuthenticationProvider"
```

```
      class="org.springframework.security.cas.authentication.CasAuthenticationProvider">
    <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.

22.1 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 its 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. It 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

22.2 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.

22.3 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.

23.1 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

23.2 Configuration

A `RunAsManager` interface is provided by Spring Security:

```
Authentication buildRunAs(Authentication authentication, Object object,
    List<ConfigAttribute> 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.access.intercept.RunAsManagerImpl">
  <property name="key" value="my_run_as_password"/>
</bean>

<bean id="runAsAuthenticationProvider"
    class="org.springframework.security.access.intercept.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

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` (`JdbcDaoImpl`) 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);
```

Group Authorities

Spring Security 2.0 introduced support for group authorities in `JdbcDaoImpl`. The table structure if groups are enabled is as follows:

```
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

There are four tables used by the Spring Security ACL implementation.

1. `acl_sid` stores the security identities recognised by the ACL system. These can be unique principals or authorities which may apply to multiple principals.

2. `acl_class` defines the domain object types to which ACLs apply. The `class` column stores the Java class name of the object.

3. `acl_object_identity` stores the object identity definitions of specific domai objects.

4. `acl_entry` stores the ACL permissions which apply to a specific object identity and security identity.

It is assumed that the database will auto-generate the primary keys for each of the identities. The `JdbcMutableAclService` has to be able to retrieve these when it has created a new row in the `acl_sid` or `acl_class` tables. It has two properties which define the SQL needed to retrieve these values `classIdentityQuery` and `sidIdentityQuery`. Both of these default to `call identity()`

Hypersonic SQL

The default schema works with the embedded HSQLDB database that is used in unit tests within the framework.

```
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 not null,
  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) );
```

PostgreSQL

```
create table acl_sid(
  id bigserial not null primary key,
  principal boolean not null,
  sid varchar(100) not null,
  constraint unique_uk_1 unique(sid,principal));

create table acl_class(
  id bigserial not null primary key,
  class varchar(100) not null,
  constraint unique_uk_2 unique(class));

create table acl_object_identity(
  id bigserial 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 bigserial 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));
```

You will have to set the `classIdentityQuery` and `sidIdentityQuery` properties of `JdbcMutableAclService` to the following values, respectively:

- `select currval(pg_get_serial_sequence('acl_class', 'id'))`
- `select currval(pg_get_serial_sequence('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. The namespace is written in RELAX NG [http://www.relaxng.org/] Compact format and later converted into an XSD schema. If you are familiar with this format, you may wish to examine the schema file [http://git.springsource.org/spring-security/spring-security/blobs/3.0.x/config/src/main/resources/org/springframework/security/config/spring-security-3.0.4.rnc] directly.

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 `SecurityContextPersistenceFilter`, an `ExceptionTranslationFilter` and a `FilterSecurityInterceptor`. These are fixed and cannot be replaced with alternatives.

<http> Attributes

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

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".

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

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`.

lowercase-comparisons

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

realm

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

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.

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`.

access-denied-page

Deprecated in favour of the `access-denied-handler` child element.

once-per-request

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

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 `SecurityContextPersistenceFilter`. `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 `SecurityContextPersistenceFilter`.

use-expressions

Enables EL-expressions in the `access` attribute, as described in the chapter on expression-based access-control.

disable-url-rewriting

Prevents session IDs from being appended to URLs in the application. Clients must use cookies if this attribute is set to `true`.

<access-denied-handler>

This element allows you to set the `errorPage` property for the default `AccessDeniedHandler` used by the `ExceptionTranslationFilter`, (using the `error-page` attribute, or to supply your own implementation using the `ref` attribute. This is discussed in more detail in the section on the `ExceptionTranslationFilter`.

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 `FilterInvocationSecurityMetadataSource` 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 `ChannelAuthenticationFilter` if particular URLs need to be accessed by HTTPS, for example. When matching the specified patterns against an incoming request, the matching is done in the order in which the elements are declared. So the most specific matches patterns should come first and the most general should come last.

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.

method

The HTTP Method which will be used in combination with the pattern to match an incoming request. If omitted, any method will match. If an identical pattern is specified with and without a method, the method-specific match will take precedence.

access

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

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

`ChannelAuthenticationFilter` will be added to the filter stack and its additional dependencies added to the application context.

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.

`filters`

Can only take the value “none”. This will cause any matching request to bypass the Spring Security filter chain entirely. None of the rest of the `<http>` configuration will have any effect on the request and there will be no security context available for its duration. Access to secured methods during the request will fail.

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.

The `<form-login>` Element

Used to add an `UsernamePasswordAuthenticationFilter` to the filter stack and an `LoginUrlAuthenticationEntryPoint` 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.

`login-page`

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

`login-processing-url`

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

`default-target-url`

Maps to the `defaultTargetUrl` property of `UsernamePasswordAuthenticationFilter`. If not set, the default value is "/" (the application root). A user will be taken to this URL after logging

[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.

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.

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 `UsernamePasswordAuthenticationFilter`. Default value is "false".

authentication-failure-url

Maps to the `authenticationFailureUrl` property of `UsernamePasswordAuthenticationFilter`. 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.

authentication-success-handler-ref

This can be used as an alternative to `default-target-url` and `always-use-default-target`, giving you full control over the navigation flow after a successful authentication. The value should be he name of an `AuthenticationSuccessHandler` bean in the application context.

authentication-failure-handler-ref

Can be used as an alternative to `authentication-failure-url`, giving you full control over the navigation flow after an authentication failure. The value should be he name of an `AuthenticationFailureHandler` bean in the application context.

The <http-basic> Element

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

The <remember-me> Element

Adds the `RememberMeAuthenticationFilter` 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.

data-source-ref

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

token-repository-ref

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

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.

token-repository-ref

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

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].

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.

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.

The <session-management> Element

Session-management related functionality is implemented by the addition of a `SessionManagementFilter` to the filter stack.

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 session fixation protection is enabled, the `SessionManagementFilter` is injected with an appropriately configured `DefaultSessionAuthenticationStrategy`. See the Javadoc for this class for more details.

The <concurrency-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

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

be created, and a `ConcurrentSessionControlStrategy` will be used with the `SessionManagementFilter`. If a `form-login` element has been declared, the strategy object will also be injected into the created authentication filter. An instance of `SessionRegistry` (a `SessionRegistryImpl` instance unless the user wishes to use a custom bean) will be created for use by the strategy.

The `max-sessions` attribute

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

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.

The `error-if-maximum-exceeded` attribute

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

The `session-registry-alias` and `session-registry-ref` attributes

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

The <anonymous> Element

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

The <x509> Element

Adds support for X.509 authentication. An `X509AuthenticationFilter` will be added to the stack and an `Http403ForbiddenEntryPoint` 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`.

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`).

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.

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 `OpenIDAuthenticationFilter` 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.

The `<logout>` Element

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

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".

The `logout-success-url` attribute

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

The `invalidate-session` attribute

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

The `<custom-filter>` Element

This element is used to add a filter to the filter chain. It doesn't create any additional beans but is used to select a bean of type `javax.servlet.Filter` which is already defined in the appllication context and add that at a particular position in the filter chain maintained by Spring Security. Full details can be found in the namespace chapter.

The `request-cache` Element

Sets the `RequestCache` instance which will be used by the `ExceptionTranslationFilter` to store request information before invoking an `AuthenticationEntryPoint`.

The `<http-firewall>` Element

This is a top-level element which can be used to inject a custom implementation of `HttpFirewall` into the `FilterChainProxy` created by the namespace. The default implementation should be suitable for most applications.

B.2 Authentication Services

Before Spring Security 3.0, an `AuthenticationManager` was automatically registered internally. Now you must register one explicitly using the `<authentication-manager>` element. This creates 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 `authentication-provider` element.

The `<authentication-manager>` Element

Every Spring Security application which uses the namespace must have include this element somewhere. It is responsible for registering the `AuthenticationManager` which provides authentication services to the application. It also allows you to define an alias name for the internal instance for use in your own configuration. Its use is described in the namespace introduction. All elements which create `AuthenticationProvider` instances should be children of this element.

The element also exposes an `erase-credentials` attribute which maps to the `eraseCredentialsAfterAuthentication` property of the `ProviderManager`. This is discussed in the Core Services chapter.

The `<authentication-provider>` Element

Unless used with a `ref` attribute, this element is shorthand 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.

The `<password-encoder>` Element

Authentication providers can optionally be configured to use a password encoder as described in the namespace introduction. This will result in the bean being injected with the appropriate `PasswordEncoder` instance, potentially with an accompanying `SaltSource` bean to provide salt values for hashing.

Using `<authentication-provider>` to refer to an `AuthenticationProvider` Bean

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:

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

B.3 Method Security

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 the section called "`access-decision-manager-ref`", using the same attribute.

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.

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.

The <after-invocation-provider> Element

This element can be used to decorate an `AfterInvocationProvider` for use by the security interceptor maintained by the `<global-method-security>` namespace. You can define zero or more of these within the `global-method-security` element, each with a `ref` attribute pointing to an `AfterInvocationProvider` bean instance within your application context.

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.

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.

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`. As with all namespace authentication providers, it must be included as a child of the `authentication-provider` element.

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`.

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.

`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_".

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.

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>`.