Spring Security Architecture & Design.

Friday, June 28, 2019

3:24 PM

Key Components of Spring Security

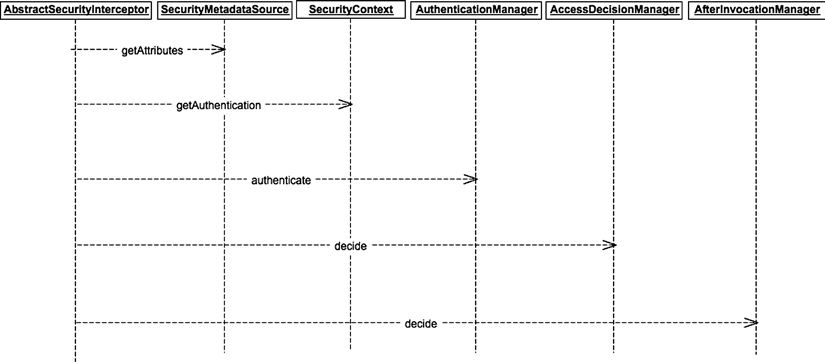
* + XML Namespace
  + Namespace Servlet Filters
  + AuthenticationManager
  + AuthenticationProvider
  + UserDetailsService
  + UserDetails
  + SecurityContexHolder
  + SecurityInterceptor
  + ACL Access Control List
  + Authentication
  + AccessDecisionManager
  + ConfigAttribute

The Security Interceptor

One of the most important components of the framework is the Security Interceptor. With the main logic implemented in AbstractSecurityInterceptor and with two concrete implementations in the form of FilterSecurityInterceptor and MethodSecurityInterceptor , the Security Interceptor is in charge of deciding whether a particular petition should be allowed to go through to a secured resource.

SecurityContextHOIder 
getContext() 
Methodlnterceptor 
invoke(MethodInv0cation) 
AbstractSecuritylnterce r 
abstract ObtainSecurityMetadataS0urce() 
AccessDecis ion Manager 
AfterlnvocationManager 
AuthenticationMa nager 
RunAsManager 
Filter 
d0FiIter ( ServIetRequest, ServtetResponse, FilterChain) 
MethodSecurityInterceptor 
FilterSecuritylnterceptor 

The Security Interceptor lies at the core of the Spring Security framework. Every call to a secured resource in Spring Security passes through this interceptor.



Note:

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

The XML Namespace

To make Spring aware of a new namespace is really simple. (That’s not to say it is simple to actually parse the information of the XML and convert it to beans—this depends on the complexity of

your DSL.) All you need is the following:

* + An xsd file defining your particular XML structure.
  + A spring.schemas file where you specify the mapping between a URL-based schema location

and the location of your xsd file in your classpath.

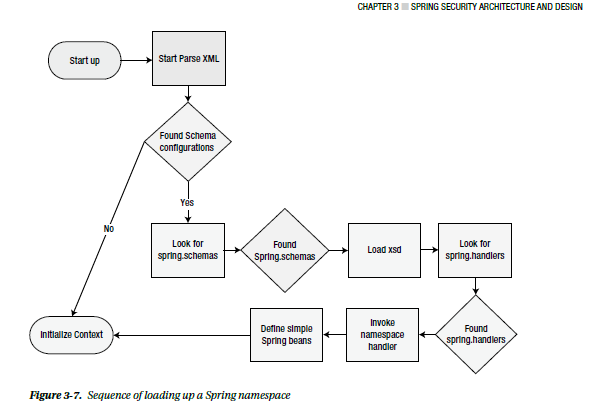
* + A spring.handlers file where you specify which class is in charge of handling everything related to your namespace
  + A bunch of parser classes that will be in charge of parsing each of the top elements defined in

your XML file

Spring Security’s config module file structure.

spring- security-config 
src/main Lava 
orgspringframework.security.config 
authentication 
debug 
http 
Idap 
method 
u:] Beanlds.java 
u:] DebugBeanDefinitionParser.java 
u] Elements.java 
u:] package-info-java 
SecurityNamespaceHandIer.java 
src/main/resources 
V orgspringframework.security.config 
catalog.xml 
-2.0.L.xsd 
pring—security 
-2.0.2.xsd 
pnng—security 
-2.0.4.xsd 
pnng—security 
-2.0.xsd 
pnng—security 
-3.0.3.xsd 
pnng—security 
-3.0.xsd 
pnng—security 
spring—security-3.I.rnc 
pnng—security-3.I.xsd 
spring 
—security.xsl 
META-INF 
spring.handlers 
spring.schemas 

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



The Filters and Filter Chain:

The filter chain model is what Spring Security uses to secure web applications. This model is built on top of the standard servlet filter functionality.

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

Defined Filters List.

* + CHANNEL\_FILTER. This filter ensures that the request is handled by the correct channel meaning, in most cases, it determines whether or not the request is handled by HTTPS.

* + CONCURRENT\_SESSION\_FILTER. This filter is part of the concurrent session-handling

mechanism. Its main function is to query the session to see if it has expired (which happens

mainly when the maximum number of concurrent sessions per user are reached) and to log

out the user if that is the case.

* + SECURITY\_CONTEXT\_FILTER. This filter populates SecurityContextHolder with a new or

existing security context to be used by the rest of the framework.

LOGOUT\_FILTER. This filter is based, by default, on a particular URL invocation

(/j\_spring\_security\_logout). It takes care of the logout process, including tasks such as

clearing the cookies, removing the remember-me information, and clearing the security context.

* + X509\_FILTER. This filter extracts the principal and credentials from an X509 certificate using

the class java.security.cert.X509Certificate and attempts to authenticate with these

pre-authenticated values.

* + PRE\_AUTH\_FILTER. This filter is used with the J2EE authentication mechanism. The J2EE

authenticated principal will be used as the pre-authenticated principal in the framework.

* + FORM\_LOGIN\_FILTER. This filter is used when a user name and password is required on a login

form. This filter takes care of authenticating with the requested user name and password. It

handles requests to a particular URL (/j\_spring\_security\_check) with a particular set of

user-name and password parameters (j\_username, j\_password).

* + LOGIN\_PAGE\_FILTER. This filter generates a default login page when the user doesn’t provide a

custom one. It will be activated when the URL /spring\_security\_login is requested.

* + BASIC\_AUTH\_FILTER. This filter processes the BASIC authentication headers in an HTTP

request. It looks for the header Authorization and tries to authenticate with these credentials.

* + REQUEST\_CACHE\_FILTER. This filter retrieves a request from the request-cache that matches the current request, and it sends the cached one through the rest of the filter chain.

* + REMEMBER\_ME\_FILTER. If no user is logged in, this filter will look to see whether there is any

“remember me” functionality active and any “remember me” Authentication available. If

there is, this filter will try to login automatically and authenticate with this “remember me”

information.

* + SESSION\_MANAGEMENT\_FILTER. This filter passes the Authentication object that

corresponds to the authenticated user who is logged in to the system to some configured

session management processors in order to do session-related handling of the

Authentication. Mainly, these processors will do some kind of validation and throw

SessionAuthenticationException if appropriate. Currently, these processors (or strategies)

include only one main class in the form of org.springframework.security.web.

authentication.session.ConcurrentSessionControlStrategy, dealing with both session

fixation and concurrent sessions.

ConfigAttribute.

The interface org.springframework.security.access.ConfigAttribute encapsulates the access information metadata present in a secured resource. For example, for our study purposes, ROLE\_ADMIN is a ConfigAttribute.

evlquuvuoneoonunsod 
OJnqwuv6uu00u01ssøudxaqaM 
ainquuvßguoopotuawpaseguolssædxapensqv 
einquuv136 
mnquuvßguoo 
0inqpuvuone00Au10Jd 

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

When you use the element <security:intercept-url pattern="/x" access="ROLE\_XX" />, Spring Security

uses the class FilterInvocationSecurityMetadataSourceParser to parse this XML. In the parsing process, the private

method parseInterceptUrlsForFilterInvocationRequestMap will be invoked. This method maps the information contained in each of the URL patterns in the XML element into a map of Ant-style request paths, like /\*, ROLE\_USER. Here /\* is an Ant pattern and ROLE\_USER is a config attribute (this says basically this config attribute is needed to access any URL with this pattern). This map, ultimately, will be set up in an instance of an implementation of the interface

org.springframework.security.web.access.intercept.FilterInvocationSecurityMetadataSource inside the FilterSecurityInterceptor, which uses it when each request comes to match the requested URL against the keys in the map to find out if the URL is secured and then extracts the ConfigAttributes against which to check the authorities of the requesting Authentication object.

Spring config loading 
Instantiate 
FilterlnvocationSecurityMetadataSourceParser 
Filtersecuritylnterceptor 
parse 
Defines 
Intercept URL XML 
Pattern: mello 
Access: ROLE_ADMIN 
Map<RequestMatcher. SecurityConfiq> 
FilterlnvocationSecurityMetadataSource 

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

the security application context XML file:

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

The Authentication Object

The Authentication object is an abstraction that represents the entity that logs in to the system most likely, a user.

*Authentication hierarchy:*

Authentication 
RunAsUserT0ken 
PreAuthenticatedAuthenticationToken 
UsernamePasswordAuthenticationT0ken 
OpenlDAuthenticationT0ken 

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

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

*The Authentication interface.*

package org.springframework.security.core;

import java.io.Serializable;

import java.security.Principal;

import java.util.Collection;

import org.springframework.security.authentication.AuthenticationManager;

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

public interface Authentication extends Principal, Serializable {

Collection<? extends GrantedAuthority> getAuthorities();

Object getCredentials();

Object getDetails();

Object getPrincipal();

boolean isAuthenticated();

void setAuthenticated(boolean isAuthenticated) throws

IllegalArgumentException;

}

Few implementations of Authentication in the framework:

* + UsernamePasswordAuthenticationToken. This is a simple implementation that contains, as its

name clearly specifies, the user name and password information of the authenticated (or pending authentication) user. It is the most common Authentication implementation used throughout

the system, as many of the AuthenticationProvider objects depend directly on this class.

* + PreAuthenticatedAuthenticationToken. This implementation exists for handling

pre-authenticated Authentication objects. Pre-authenticated authentications are those

where the actual authentication process is handled by an external system, and Spring

Security deals only with extracting the principal (or user) information out of the external system's messages.

* + OpenIDAuthenticationToken. This is an Authentication implementation used specifically

for OpenID authentication schemes. It is used by both the OpenID filter and the OpenID

authentication provider.

* + RunAsUserToken. This implementation is used by the RunAsManager, which is called by the

Security Interceptor, when the accessed resource contains a ConfigAttribute that starts with

the prefix 'RUN\_AS\_'. If there is a ConfigAttribute with this value, RunAsManager adds new

GrantedAuthorities to the authenticated user corresponding to the RUN\_AS value.

SecurityContext and SecurityContextHolder

The interface org.springframework.security.core.context.SecurityContext (actually, its implementation is SecurityContextImpl) is the place where Spring Security stores the valid Authentication object, associating it with the current thread. The org.springframework.security.core.context.SecurityContextHolder is the class

used to access SecurityContext from many parts of the framework. It is built mainly of static methods to store and access SecurityContext, delegating to configurable strategies the way to handle this SecurityContext—for example, one SecurityContext per thread (default), one global SecurityContext, or a custom strategy.

1 du naos 
KBełensłep10HłxaauooĂwnoas 

AuthenticationProvider

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

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

* + CasAuthenticationProvider
  + JaasAuthenticationProvider
  + DaoAuthenticationProvider
  + OpenIDAuthenticationProvider
  + RememberMeAuthenticationProvider
  + LdapAuthenticationProvider

AccessDecisionManager

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

apply a determined strategy on whether or not to grant access. Voters, however, also can abstain from voting.

AffirmativeBased 
AccessDecisionManager 
ConsensusBased 
UnanimousBased 

*AffirmativeBased*

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

*ConsensusBased*

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

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

AccessDecisionVoter

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

The main method is “vote”, and as can be deduced from the interface, it will return one of three possible responses (ACCESS\_GRANTED, ACCES\_ABSTAIN, ACCESS\_DENIED), depending on whether the required conditions are satisfied. The satisfaction or not of the conditions is given by analyzing the Authentication object’s rights against the required resource. In practice this basically means that the Authentication’s authorities are compared against the resource’s security attributes looking for matches.

Following are the current AccessDecisionVoter implementations:

* + org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter.

This voter votes on resources with expression configurations based on @PreFilter and

@PreAuthorize annotations. @PreFilter and @PreAuthorize annotations support a value

attribute that can have a SpEL expression. The PreInvocationAuthorizationAdviceVoter

is the one in charge of evaluating the SpEL expressions (of course with the help of Spring’s SpEL

evaluation mechanism) provided in these annotations. We will be explaining and using SpEL

expressions in several parts of the book so this concept will become clearer as the books advances.

* + org.springframework.security.access.vote.RoleVoter. This is, perhaps, the most

commonly used voter of them all. This voter, by default, is able to vote on resources

that have ConfigAttribute(s) containing security metadata starting with the prefix

“ROLE\_” (which can be overridden). When an Authentication object tries to access the

resource, its GrantedAuthorities will be matched against the relevant ConfigAttributes.

If there is a match, access is granted. If there isn’t, access is denied.

* + org.springframework.security.access.expression.WebExpressionVoter. This is the

voter in charge of evaluating SpEL expressions in the context of web requests in the filter

chain—expressions like 'hasRole' in the <intercept-url> element. To make use of this voter,

and in general to support SpEL expressions in web security, the use-expressions="true"

attribute needs to be added to the <http> element.

UserDetailsService and AuthenticationUserDetailsService

The interface org.springframework.security.core.userdetails.UserDetailsService is in charge of loading the user information from the underlying user store (in-memory, database, and so on) when an authentication request arrives in the application. UserDetailsService makes use of the provided user name for looking up the rest of the required user data from the data store. It defines just one method.

*UserDetailsServicepackage org.springframework.security.core.userdetails*

public interface UserDetailsService {

UserDetails loadUserByUsername(String username) throws UsernameNotFoundException;

}

onu 1 
'06euewsııewoxosn 

The AuthenticationUserDetailsService interface.

These are the two main strategies (AuthenticationUserDetailsService and UserDetailsService) used

for retrieving the user information when attempting authentication. They are usually called from the particular AuthenticationProvider that is being used in the application. For example, the OpenIDAuthenticationProvider and CasAuthenticationProvider delegate to an AuthenticationUserDetailsService to obtain the user details, while the DaoAuthenticationProvider delegates directly to a UserDetailsService. Some other providers don’t use a user details service of any kind (for example, JaasAuthenticationProvider uses its own mechanism to retrieve the

Principal from a javax.security.auth.login.LoginContext), and some others use a completely custom one (for example, LdapAuthenticationProvider uses a UserDetailsContextMapper).

package org.springframework.security.core.userdetails;

public interface AuthenticationUserDetailsService<T extends Authentication> {

UserDetails loadUserDetails(T token) throws UsernameNotFoundException;

}

UserDetails

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

can be obtained by calling the getPrincipal method on it.

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

*UserDetails*

package org.springframework.security.core.userdetails;

import org.springframework.security.core.Authentication;

import org.springframework.security.core.GrantedAuthority;

import java.io.Serializable;

import java.util.Collection;

public interface UserDetails extends Serializable {

Collection<? extends GrantedAuthority> getAuthorities();

String getPassword();

String getUsername();

boolean isAccountNonExpired();

boolean isAccountNonLocked();

boolean isCredentialsNonExpired();

boolean isEnabled();

}

ACL

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

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