M. Nardone, C. Scarioni, *Pro Spring Security*
https://doi-org.ezproxy.sfpl.org/10.1007/979-8-8688-0035-1_4

4. Spring Security Architecture and Design

Massimo Nardone[1] and Carlo Scarioni[2]
(1) HELSINKI, Finland
(2) Surbiton, UK

In Chapter 3, you developed an initial application secured with Spring Security. You got an overview of how this application worked and looked at some of the Spring Security components put into action in common Spring Security–secured applications. This chapter extends those explanations and delve deeply into the framework.

We'll look at the main components of the framework, explain the work of the servlet filters for securing web applications, look at how Spring aspect-oriented programming helps you unobtrusively add security, and, in general, show how the framework is designed internally.

What Components Make up Spring Security?

This section looks at the major components that make Spring Security work. It presents a big-picture framework overview and then delves deeper into each major component.

The 10,000-Foot View

Spring Security is a relatively flexible framework that aims to make it easy for the developer to implement security in an application. At the most general level, it's a framework composed of intercepting rules for granting or not granting access to resources. Figure 4-1 illustrates this.

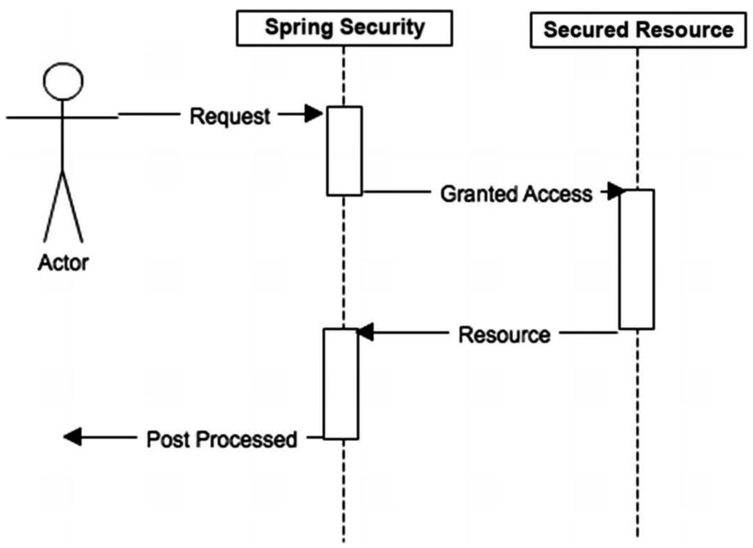


Figure 4-1 Spring Security 10,000-foot overview

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

The 1,000-Foot View

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

Spring Security's interception security model applies to two main areas of your application: URLs and method invocations. Spring Security wraps around these two *entry points* of your application and allows access only when the security constraints are satisfied. Both the method call and the filter-based security depend on a central *security interceptor,* where the main logic resides to decide whether access should be granted. Figure 4-2 shows a detailed overview of the framework.

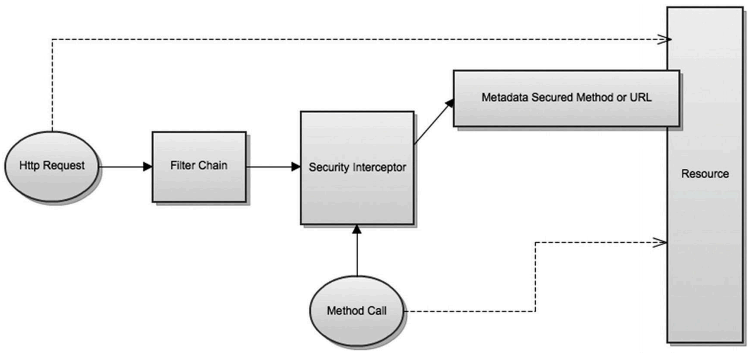


Figure 4-2 In this view, both method calls and HTTP requests try to access a resource, but first they must go through the Security Interceptor

The 100-Foot View

Spring Security might seem simple conceptually, but a lot is happening internally in a very well-built software tool. This next overview shows the main collaborating parts that enforce your security constraints. This is particularly achievable with an open source project like Spring Security, which allows you to get into the framework and appreciate its design and architecture by directly accessing the source code. After that, we'll delve deeper into the implementation details.

What follows is the best way to understand Spring Security from the inside. The enumeration of what we consider to be the main components of the framework helps you know where everything belongs and how your application is enforcing the security rules that you specify for it.

The most important Spring Security internal architecture core modules are

- Authentication
- Authorization

The process of the Authentication and Authorization modules were introduced in Chapter 1.

Figure 4-3 illustrates all the concepts/components.

XML Namespace
Servlet Filters
AuthenticationManager
AuthenticationProvider
UserDetailsService
UserDetails
SecurityContextHolder
SecurityInterceptor
AccessDecisionManager
Authentication
ConfigAttribute
ACL
Taglib

Figure 4-3 The key components of Spring Security

The Security Interceptor

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

The Security Interceptor works with a preprocessing step and a postprocessing step. The preprocessing step looks to see whether the requested resource is secured with some metadata information (or `ConfigAttribute`). If it is not, the request is allowed to continue its way either to the requested URL or method. If the requested resource is secured, the Security Interceptor retrieves the `Authentication` object from the current `SecurityContext`. If necessary, the `Authentication` object is authenticated against the configured `AuthenticationManager` with the following method.

```
public interface AuthenticationManager {

  Authentication authenticate(Authentication authentication)

    throws AuthenticationException;
}
```

An `AuthenticationManager` can do mainly three things with its method.

- Return an Authentication with value `authenticated=true` if the input represents a valid principal and can be verified
- Throw an `AuthenticationException` if the input represents an invalid principal
- Return null if it can't decide

`ProviderManager` (which delegates to a chain of `AuthenticationProvider` instances) is the most commonly used implementation of `AuthenticationManager`.

ProviderManager is the most commonly used implementation of AuthenticationManager. ProviderManager delegates to a List of AuthenticationProvider instances. Each AuthenticationProvider has an opportunity to indicate that authentication should be successful, fail, or indicate it cannot make a decision and allow a downstream AuthenticationProvider to decide. If none of the configured AuthenticationProvider instances can authenticate, authentication fails with a ProviderNotFoundException, which is a special AuthenticationException that indicates that the ProviderManager was not configured to support the type of authentication that was passed into it.

An example of the `AuthenticationManager` hierarchy using `ProviderManager` is shown in Figure 4-4.

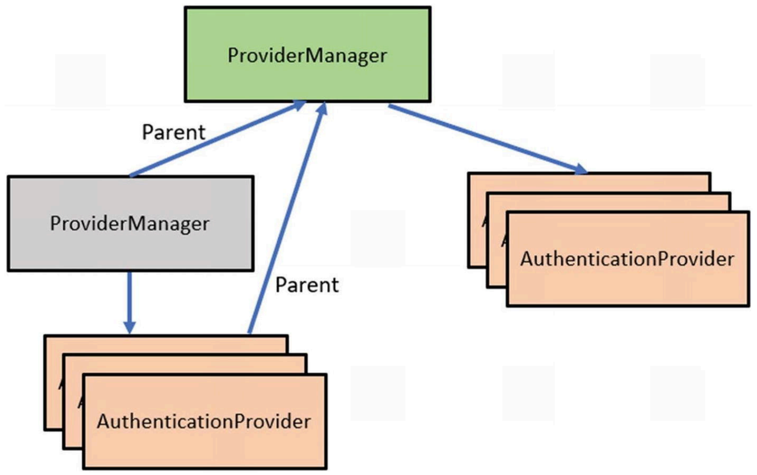


Figure 4-4 AuthenticationManager hierarchy using ProviderManager

After the object is authenticated, `AccessDecisionManager` is called to determine whether the authenticated entity can finally access the resource. `AccessDecisionManager` throws an `AccessDeniedException` if the authenticated entity cannot access the resource. If `AccessDecisionManager` decides that the `Authentication` entity is allowed to access the resource, the `Authentication` object is passed to `RunAsManager` if this is configured. If `RunAsManager` is not configured, a no-op implementation is called. `RunAsManager` returns either null (if it's not configured to be used) or a new `Authentication` object containing the same principal, credentials, and granted authorities as the original `Authentication` object, plus a new set of authorities based on the `RUN_AS` that is being used. This new `Authentication` object is put into the current `SecurityContext`.

After this processing, and independently of whether or not a `RUN_AS` `Authentication` object is used, the Security Interceptor creates a new `InterceptorStatusToken` with information about the `SecurityContext` and the `ConfigAttributes`. This token is used later in the postprocessing step of the Security Interceptor. At this point, the Security Interceptor is ready to allow access to the secured resource, so it passes the invocation through, and the particular secured entity (either a URL or a method) is invoked. After the invocation returns, the second phase of the Security Interceptor comes into play, and the postprocessing begins. The postprocessing step is considerably simpler and involves only calling an AfterInvocationManager's decide method if one is configured. In its current implementation, `AfterInvocationManager` delegates to instances of `PostInvocationAuthorizationAdvice`, which ultimately filters the returned objects or throws an `AccessDeniedException` if necessary. This is the case if you use the post-invocation filters in method-level security, as discussed in Chapter 5. In the case of web security, the `AfterInvocationManager` is null.

That is a lot of work for the security interceptor. However, because the framework is nicely modular at the class level, you can see that the Security Interceptor simply delegates most of the task to a series of well-defined collaborators; under the single-responsibility principle (SRP), it focuses on narrowly scoped responsibilities. This is a good software design and an example you should emulate. As shown in Listing 4-1, you paste the main parts of the code from `AbstractSecurityInterceptor` so that you can see the things we've been talking about. Comments are included in the code so that you can understand better what it does; they start with `//----`.

The AbstractSecurityInterceptor calls AccessDecisionManager, which is responsible for making final access control decisions, and it contains three methods.

```
void decide(Authentication authentication, Object secureObject,

      Collection<ConfigAttribute> attrs) throws AccessDeniedException;
```

```
boolean supports(ConfigAttribute attribute);



boolean supports(Class clazz);
```

The Spring Security 6 class is at https://docs.spring.io/spring-security/site/docs/current/api/org/springframework/security/access/intercept/AbstractSecurityInte

The entire `AbstractSecurityInterceptor` course code is on GitHub at https://github.com/spring-projects/spring-security/blob/master/core/src/main/java/org/springframework/security/access/intercept/AbstractSec

```
protected InterceptorStatusToken beforeInvocation(Object object) {

Assert.notNull(object, "Object was null");

final boolean debug = logger.isDebugEnabled();

// --- Here we are checking if this filter is able to process a particular type of object. For example

if (!getSecureObjectClass().isAssignableFrom(object.getClass())) {

throw new IllegalArgumentException("Security invocation attempted for object "

                    + object.getClass().getName()

                    + " but AbstractSecurityInterceptor only configured to support secure objects of ty

                    + getSecureObjectClass());

        }

// ---- Here we are retrieving the security metadata that maps to the object we are receiving. So if we

        Collection<ConfigAttribute> attributes = this.obtainSecurityMetadataSource().getAttributes(obje
if (attributes == null || attributes.isEmpty()) {

if (rejectPublicInvocations) {

throw new IllegalArgumentException("Secure object invocation " + object +

" was denied as public invocations are not allowed via this interceptor. "

                                + "This indicates a configuration error because the " + "rejectPublicIn

            }

if (debug) {

logger.debug("Public object - authentication not attempted");

            }

publishEvent(new PublicInvocationEvent(object));

return null; // no further work post-invocation

        }

if (debug) {

logger.debug("Secure object: " + object + "; Attributes: " + attributes);

        }

if (SecurityContextHolder.getContext().getAuthentication() == null) {

credentialsNotFound(messages.getMessage("AbstractSecurityInterceptor.authenticationNotFound",

                    "An Authentication object was not found in the SecurityContext"), object, attribute

        }

        Authentication authenticated = authenticateIfRequired();
```

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

try {

this.accessDecisionManager.decide(authenticated, object, attributes);

        }

catch (AccessDeniedException accessDeniedException) {

publishEvent(new AuthorizationFailureEvent(object, attributes, authenticated, accessDeniedException));

throw accessDeniedException;

        }

if (debug) {

logger.debug("Authorization successful");

        }

if (publishAuthorizationSuccess) {

publishEvent(new AuthorizedEvent(object, attributes, authenticated));

        }

   // ---- Here it will try to use the run-as functionality of Spring Security that allows a user

// --to impersonate another one acquiring its security roles, or more precisely, its

//--GrantedAuthority (s)

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

if (runAs == null) {

if (debug) {
```

```
logger.debug("RunAsManager did not change Authentication object");

            }

            // no further work post-invocation

return new InterceptorStatusToken(SecurityContextHolder.getContext(), false, attributes, object);

        } else {

if (debug) {

logger.debug("Switching to RunAs Authentication: " + runAs);

            }

SecurityContext origCtx = SecurityContextHolder.getContext();

SecurityContextHolder.setContext(SecurityContextHolder.createEmptyContext());

SecurityContextHolder.getContext().setAuthentication(runAs);

            // need to revert to token.Authenticated post-invocation

return new InterceptorStatusToken(origCtx, true, attributes, object);

        }

// ---- If the method has not thrown an exception at this point, it is safe to continue

// ---- the invocation through to the resource. Authorization has been granted.

    }

protected Object afterInvocation(InterceptorStatusToken token, Object returnedObject) {

if (token == null) {

            // public object
```

```
return returnedObject;

        }

if (token.isContextHolderRefreshRequired()) {

if (logger.isDebugEnabled()) {

logger.debug("Reverting to original Authentication: " + token.getSecurityContext().getAuthentication())

            }

SecurityContextHolder.setContext(token.getSecurityContext());

        }

// ---- If there is an afterInvocationManager configured, it will be called.

// ---- It will take care of filtering the return value or actually throwing an exception

//----- if it is relevant to do so.

if (afterInvocationManager != null) {

            // Attempt after invocation handling

try {

returnedObject = afterInvocationManager.decide(token.getSecurityContext().getAuthentication(),

token.getSecureObject(),

token.getAttributes(), returnedObject);

            }

catch (AccessDeniedException accessDeniedException) {

AuthorizationFailureEvent event = new AuthorizationFailureEvent(token.getSecureObject(), token .getAttr
publishEvent(event);

throw accessDeniedException;

            }

        }

// ---- Here is the full authorization cycled finished. The response is returned to the caller.

return returnedObject;

    }
```

Listing 4-1 AbstractSecurityInterceptor

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

Figure 4-5 shows the interceptor in an unified modeling language (UML) class diagram. Figure 4-6 shows a simplified sequence diagram.

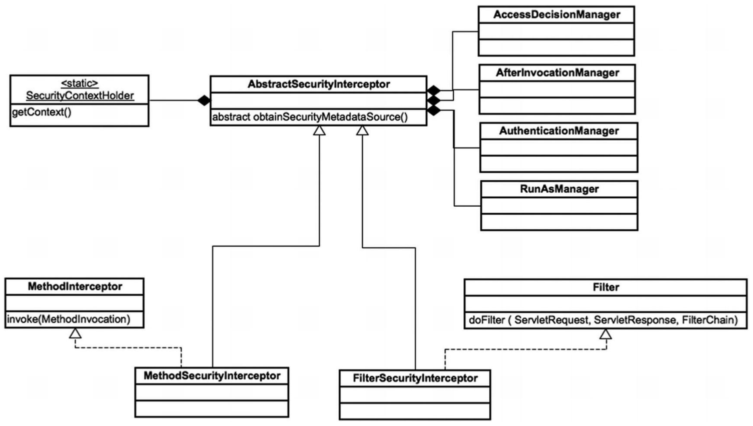

<static>
SecurityContextHolder
getContext()
AbstractSecurityInterceptor
abstract obtainSecurityMetadataSource()
AccessDecisionManager
AfterInvocationManager
AuthenticationManager
RunAsManager
MethodInterceptor
invoke(MethodInvocation)
Filter
doFilter (ServletRequest, ServletResponse, FilterChain)
MethodSecurityInterceptor
FilterSecurityInterceptor

Figure 4-5 SecurityInterceptor UML class diagram, simplified

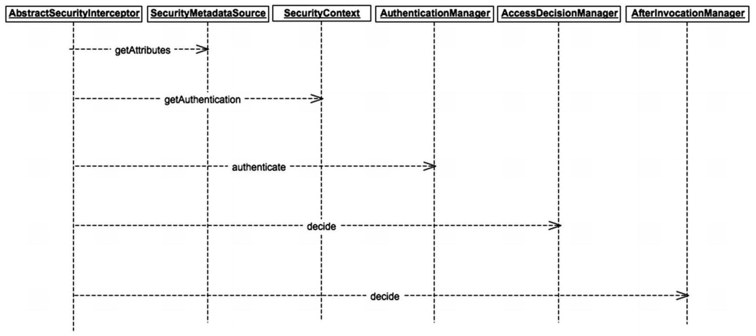


Figure 4-6 AbstractSecurityInterceptor sequence diagram, simplified

You know how security interceptors work, but how do they come to be? How do they know what to intercept? The answer lies in the next few components, so keep reading.

The XML Namespace

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

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

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

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

Spring 2.0 introduced support for defining custom XML namespaces. Since then, many projects have used this facility, making them more attractive to work with.

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

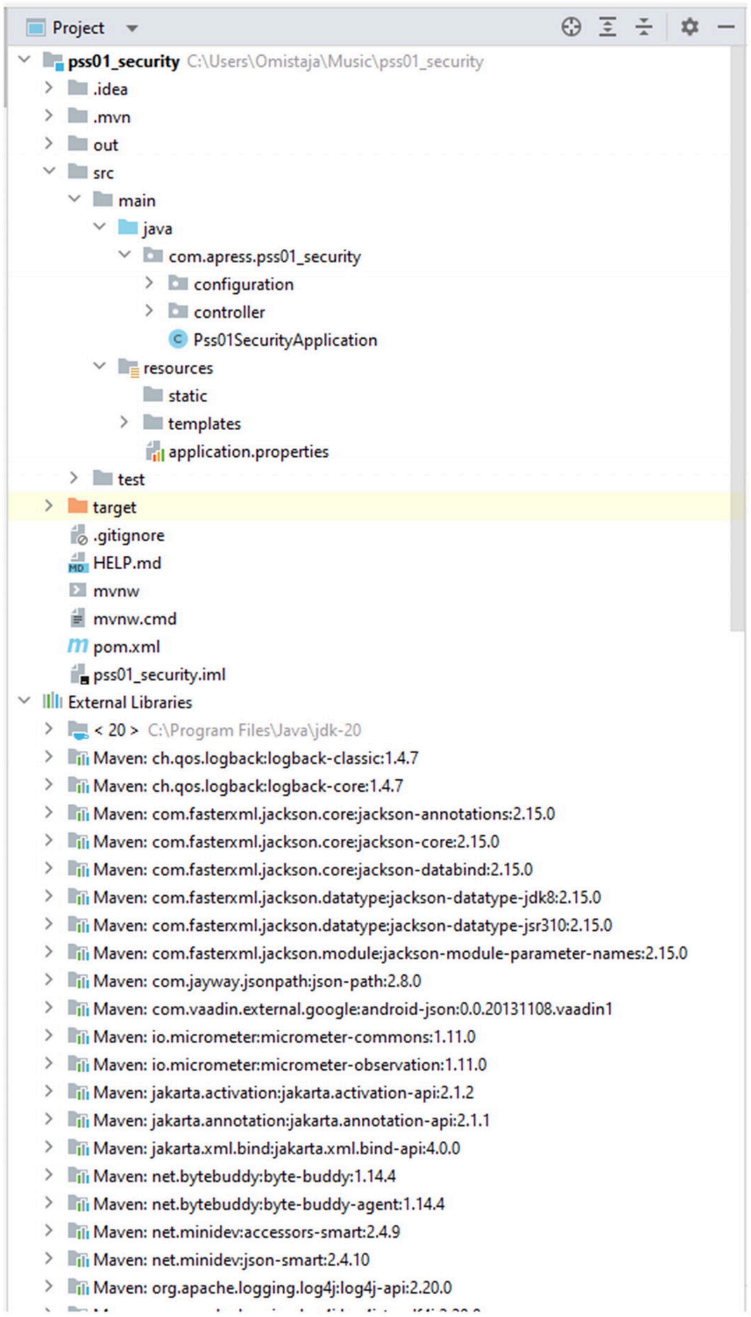
Note A DSL is a language customized to represent the concepts of a particular application domain. Sometimes, a whole new language is created to support the new domain, which is referred to as an *external DSL*. An existing language is sometimes tweaked to allow for new expressions that represent the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

Making Spring aware of a new namespace is simple. (That's not to say it is simple to parse the XML's information 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
- Several parser classes that parse each of the top elements are defined in your XML file.

Chapter 8 provides some examples of how to create a new namespace element and integrate it with Spring Security.

All the namespace configuration-related information resides in the config module for Spring Security. In Figure 4-7, you can see the expanded structure of the `config` module as seen in the IntelliJ IDEA 2023.1.2 integrated development environment (IDE) used in this book.



```
Project
pss01_security C:\Users\Omistaja\Music\pss01_security
  .idea
  .mvn
  out
  src
    main
      java
        com.apress.pss01_security
          configuration
          controller
          Pss01SecurityApplication
      resources
        static
        templates
        application.properties
    test
  target
  .gitignore
  HELP.md
  mvnw
  mvnw.cmd
  pom.xml
  pss01_security.iml
External Libraries
  < 20 > C:\Program Files\Java\jdk-20
  Maven: ch.qos.logback:logback-classic:1.4.7
  Maven: ch.qos.logback:logback-core:1.4.7
  Maven: com.fasterxml.jackson.core:jackson-annotations:2.15.0
  Maven: com.fasterxml.jackson.core:jackson-core:2.15.0
  Maven: com.fasterxml.jackson.core:jackson-databind:2.15.0
  Maven: com.fasterxml.jackson.datatype:jackson-datatype-jdk8:2.15.0
  Maven: com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.15.0
  Maven: com.fasterxml.jackson.module:jackson-module-parameter-names:2.15.0
  Maven: com.jayway.jsonpath:json-path:2.8.0
  Maven: com.vaadin.external.google:android-json:0.0.20131108.vaadin1
  Maven: io.micrometer:micrometer-commons:1.11.0
  Maven: io.micrometer:micrometer-observation:1.11.0
  Maven: jakarta.activation:jakarta.activation-api:2.1.2
  Maven: jakarta.annotation:jakarta.annotation-api:2.1.1
  Maven: jakarta.xml.bind:jakarta.xml.bind-api:4.0.0
  Maven: net.bytebuddy:byte-buddy:1.14.4
  Maven: net.bytebuddy:byte-buddy-agent:1.14.4
  Maven: net.minidev:accessors-smart:2.4.9
  Maven: net.minidev:json-smart:2.4.10
  Maven: org.apache.logging.log4j:log4j-api:2.20.0
```

Figure 4-7 Spring Security's file structure

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

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

Let's suppose you create and run a Spring-based application using XML-defined application context configuration with some Spring Security namespace definitions. When it starts to load, it looks in the application context's namespace definitions at the top of the XML configuration file. It finds the reference to the Spring Security namespace (normally a reference like `xmlns:security="http://www.springframework.org/schema/security"`). Using the information from the `spring.handlers` mapping file, it sees that the file to handle the security elements is the final class, `org.springframework.security.config.SecurityNamespaceHandler`. Spring calls the `parse` method of this class for every top element in the configuration file that uses the security namespace. Figure 4-8 shows the load-up sequence for this process.

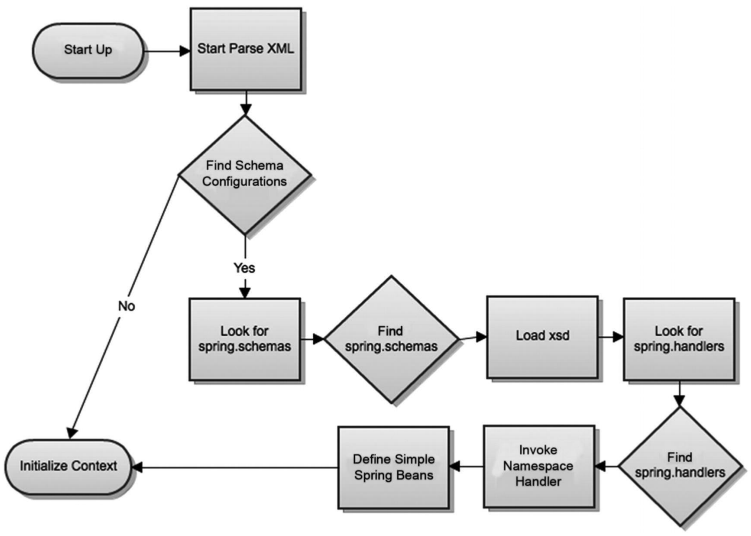


Figure 4-8 Sequence of loading up a Spring namespace

`SecurityNamespaceHandler` delegates to a series of `BeanDefinitionParser` objects for the individual parsing of each top-level element. The whole list of elements supported in the Spring Security namespace configuration is defined in the class `org.springframework.security.config.Elements` as constants. This class is shown in Listing 4-2.

```
package org.springframework.security.config;

public abstract class Elements {

public static final String ACCESS_DENIED_HANDLER = "access-denied-handler";

public static final String AUTHENTICATION_MANAGER = "authentication-manager";

public static final String AFTER_INVOCATION_PROVIDER = "after-invocation-provider";

public static final String USER_SERVICE = "user-service";

public static final String JDBC_USER_SERVICE = "jdbc-user-service";

public static final String FILTER_CHAIN_MAP = "filter-chain-map";

public static final String INTERCEPT_METHODS = "intercept-methods";

public static final String INTERCEPT_URL = "intercept-url";
public static final String AUTHENTICATION_PROVIDER = "authentication-provider";

public static final String HTTP = "http";

public static final String LDAP_PROVIDER = "ldap-authentication-provider";

public static final String LDAP_SERVER = "ldap-server";

public static final String LDAP_USER_SERVICE = "ldap-user-service";

public static final String PROTECT_POINTCUT = "protect-pointcut";

public static final String EXPRESSION_HANDLER = "expression-handler";

public static final String INVOCATION_HANDLING = "pre-post-annotation-handling";

public static final String INVOCATION_ATTRIBUTE_FACTORY = "invocation-attribute-factory";

public static final String PRE_INVOCATION_ADVICE = "pre-invocation-advice";

public static final String POST_INVOCATION_ADVICE = "post-invocation-advice";

public static final String PROTECT = "protect";

public static final String SESSION_MANAGEMENT = "session-management";

public static final String CONCURRENT_SESSIONS = "concurrency-control";

public static final String LOGOUT = "logout";

public static final String FORM_LOGIN = "form-login";

public static final String BASIC_AUTH = "http-basic";

public static final String REMEMBER_ME = "remember-me";

public static final String ANONYMOUS = "anonymous";

public static final String FILTER_CHAIN = "filter-chain";
public static final String GLOBAL_METHOD_SECURITY = "global-method-security";

public static final String PASSWORD_ENCODER = "password-encoder";

public static final String SALT_SOURCE = "salt-source";

public static final String PORT_MAPPINGS = "port-mappings";

public static final String PORT_MAPPING = "port-mapping";

public static final String CUSTOM_FILTER = "custom-filter";

public static final String REQUEST_CACHE = "request-cache";

public static final String X509 = "x509";

public static final String JEE = "jee";

public static final String FILTER_SECURITY_METADATA_SOURCE = "filter-security-metadata-source";

public static final String METHOD_SECURITY_METADATA_SOURCE = "method-security-metadata-source";

    @Deprecated

public static final String FILTER_INVOCATION_DEFINITION_SOURCE = "filter-invocation-definition-source";

public static final String LDAP_PASSWORD_COMPARE = "password-compare";

public static final String DEBUG = "debug";

public static final String HTTP_FIREWALL = "http-firewall";

}
```

Listing 4-2
Constants for All the Spring Security Namespace Elements

From the list of elements presented in the previous class, the top-level ones used in the XML configuration files are as follows. (Listing **4-2** refers to them by the constant name, not the XML element name).

- `LDAP_PROVIDER` configures your application's Lightweight Directory Access Protocol (LDAP) authentication provider in case you require one.
- `LDAP_SERVER` configures an LDAP server in your application.
- `LDAP_USER_SERVICE` configures the service for retrieving user details from an LDAP server and populating that user's authorities (Spring Security uses the term "authorities" to refer to the permission names that are granted to a particular user. For example, `ROLE_USER` is an authority).
- `USER_SERVICE` defines the in-memory user service where you can store user names, credentials, and authorities directly in the application context definition file. This type of configuration is specific for test environments and academic purposes because it is easy to set up and fast.
- `JDBC_USER_SERVICE` allows you to set up a database-driven user service, where you specify a `DataSource` and the queries to retrieve the user information from a database.
- `AUTHENTICATION_PROVIDER` defines `DaoAuthenticationProvider`, which is an authentication provider that delegates to an instance of `UserDetailsService`, which can be any of the ones defined in the previous three elements or a reference to a customized one.
- `GLOBAL_METHOD_SECURITY` sets up the global support in your application to the annotations `@Secured`, `@javax.annotation.security.RolesAllowed`, `@PreAuthorize`, and `@PostAuthorize`. This element is the one that handles the registration of a method interceptor that is aware of all the metadata of the bean's methods to apply the corresponding security advice.
- `AUTHENTICATION_MANAGER` registers a global `ProviderManager` in the application and sets up the configured `AuthenticationProviders` on it.
- `METHOD_SECURITY_METADATA_SOURCE` registers `MapBasedMethodSecurityMetadataSource` in the application context. It holds `Map<RegisteredMethod, List<ConfigAttribute>>`. It does this so that when a request is made to a method, the method can be retrieved, and its security constraints can be checked.
- `DEBUG` registers a `DebugFilter` in the security filter chain.
- `HTTP` is the main element for a web-based secure application. The HTTP element is really powerful. It allows for the definition of URL-based security-mapping strategies, the configuration of the filters, the Secure Sockets Layer (SSL) support, and other HTTP-related security configurations.
- `HTTP_FIREWALL` uses a firewall element and adds it to the filter chain if it is configured. The firewall referenced should be an implementation of Spring's own `HttpFirewall` interface.
- `FILTER_INVOCATION_DEFINITION_SOURCE` has been deprecated. See the following one.

- `FILTER_SECURITY_METADATA_SOURCE` wraps a list of `<intercept-url>` elements. These elements map the relationship between URLs and the `ConfigAttributes` required for accessing those URLs.
- `FILTER_CHAIN` allows you to configure the Spring Security filter chain used in the application, which filters you want to add to the chain, and a request matcher to customize how the chain matches requests. The most important request matches are ant-based and regexp-based.

The Spring Security namespace is used throughout the book, so many of the elements described here are revisited in later chapters.

The Filters and Filter Chain

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

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

A typical filter for a single HTTP request is shown in Figure **4-9**.

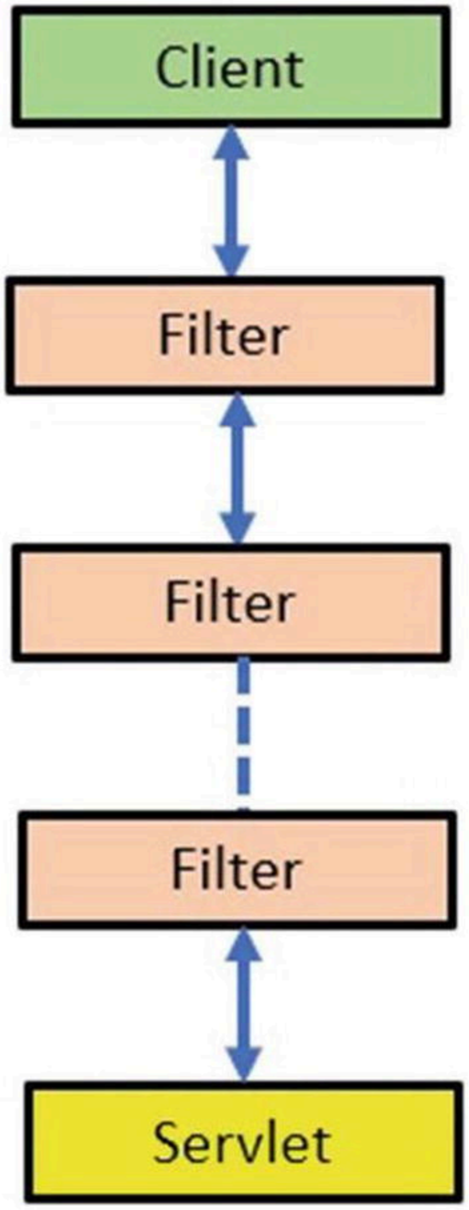


Figure 4-9 Spring Security filter example for a single HTTP request

The Spring Security filter chain is made up of Spring beans; however, standard servlet-based web applications don't know about Spring beans.

From the container's point of view, Spring Security is a single filter, which internally contains a lot of filters with different purposes.

Spring Security is installed as a single filter in the FilterChainProxy chain, which contains all the security needed, as shown in Figure **4-10**.

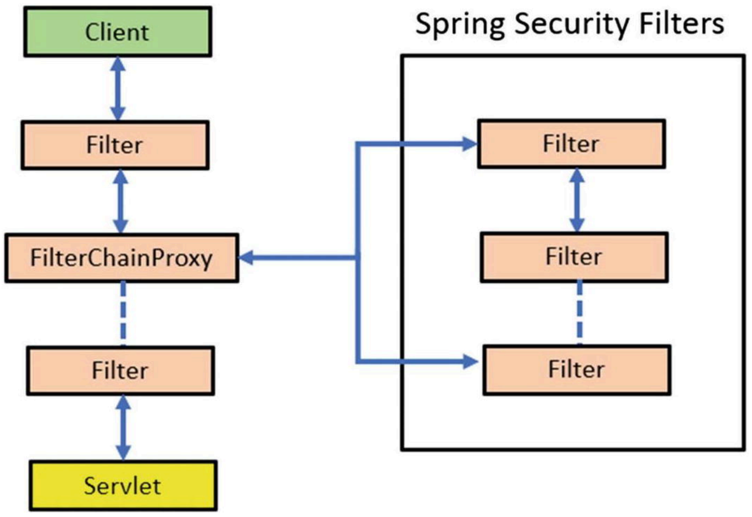


Figure 4-10 Spring Security filters overview

In the security filter, a special layer of indirection is installed in the `DelegatingFilterProxy` container, which does not need to be a Spring bean.

The flow works so that the `DelegatingFilterProxy` filter delegates to a `FilterChainProxy`, which instead is always a bean with a fixed name of `springSecurityFilterChain`, which at the end is responsible, within your application, for the following.

- Protecting the application URLs
- Validating the submitted username and passwords
- Redirecting to the login form

The `DelegatingFilterProxy` process containing the `FilterChainProxy` is shown in Figure 4-11.

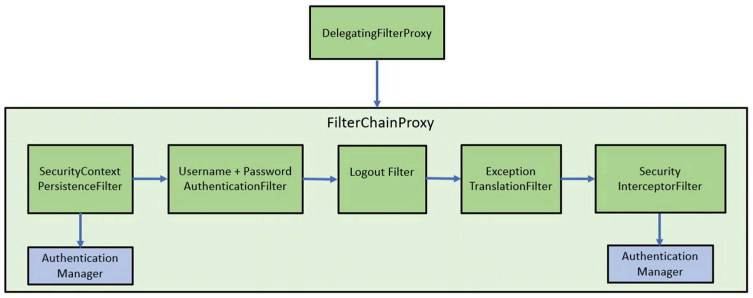

DelegatingFilterProxy
FilterChainProxy
SecurityContext PersistenceFilter
Username + Password AuthenticationFilter
Logout Filter
Exception TranslationFilter
Security InterceptorFilter
Authentication Manager
Authentication Manager

Figure 4-11 Spring Security filter chain overview

The Spring Security filter configuration is achieved via a special servlet and two main XML files, `web.xml` and `applicationContext.xml`. Starting with Servlet 3.0, `web.xml` is no longer necessary.

This special servlet filter is needed to cross the boundaries between the standard servlet API and life cycle and the Spring application where the bean filters reside. This is the job of the `org.springframework.web.filter.DelegatingFilterProxy`. It is defined in `web.xml`, which uses the `WebApplicationContextUtils.getWebApplicationContext` utility method to retrieve the root application context of the application. These two classes are from Spring Framework, not Spring Security.

Figure 4-12 shows the configuration of the filter chain.

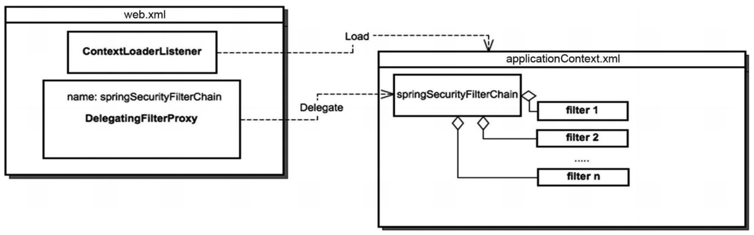


Figure 4-12 Understanding the Spring Security filter configuration. The filter in the web.xml file has the same name as the bean in the Spring application context so that the listener can find it

More information about migrating from Spring Security 6 filters is at

https://docs.spring.io/spring-security/reference/servlet/architecture.html#servlet-securityfilterchain.

The filter chain is explained in Chapter 5. However, here we'll provide an overview of the available filters and what they do. The available filters in version 6 are defined as enums in the `org.springframework.security.web.SecurityFilterChain` file .

The enums are then referenced later in the startup process when instantiating the bean definitions for each filter. The following describes the defined filters.

- `CHANNEL_FILTER` ensures that the request is handled by the correct channel—meaning, in most cases, it determines whether HTTPS handles the request.
- `CONCURRENT_SESSION_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` populates `SecurityContextHolder` with a new or existing security context to be used by the rest of the framework.
- `LOGOUT_FILTER` is based, by default, on a particular URL invocation (`/logout`). It handles the logout process, including clearing the cookies, removing the "remember me" information, and clearing the security context.
- `X509_FILTER` extracts the principal and credentials from an X509 certificate using the class `java.security.cert.X509Certificate` and attempts to authenticate with these preauthenticated values.
- `PRE_AUTH_FILTER` is used with the J2EE authentication mechanism. The J2EE authenticated principal is the preauthenticated principal in the framework.
- `FORM_LOGIN_FILTER` is used when a username and password are required on a login form. This filter takes care of authenticating with the requested username and password. It handles (since Spring 4) requests to a particular URL (`/login`) with a particular set of username and password parameters (`username`, `password`).

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

- `SWITCH_USER_FILTER` allows a user to impersonate another one by visiting a URL that has been updated from `/j_spring_security_switch_user` to `/login/impersonate` since Spring 4. This URL should be secured to allow certain users access to this functionality. Also, the method `attemptSwitchUser` in the implementing class `SwitchUserFilter` can be overridden to add constraints so that you can use more finely-grained information to decide whether certain users are allowed or not allowed to impersonate other users.

Many interfaces (e.g., Open ID and ConfigAttribute) have been deprecated since API version 5.6.2. The entire list is at **https://docs.spring.io/spring-security/site/docs/5.6.2/api/deprecated-list.html**.

The Authentication Object

In Spring Security 6, the Authentication interface serves two main purposes within Spring Security.

- An input to AuthenticationManager to provide the credentials a user has provided to authenticate. When used in this scenario, isAuthenticated() returns false.
- Represent the currently authenticated user. You can obtain the current authentication from SecurityContext.

The Authentication interface contains the following.

- **principal** identifies the user. Authenticating with a username/password is often an instance of UserDetails.
- **credentials** is usually the password. It is often cleared after the user is authenticated to ensure it is not leaked.
- **authorities** means the GrantedAuthority instances, which are high-level permissions the user is granted. Two examples are roles and scopes.

The `Authentication` object is an abstraction representing the entity that logs in to the system—most likely a user. Because it is normally a user authenticating, we'll assume so and use the term "user" in the rest of the book. There are a few implementations of the `Authentication` object in the framework, as shown in Figure **4-13**.

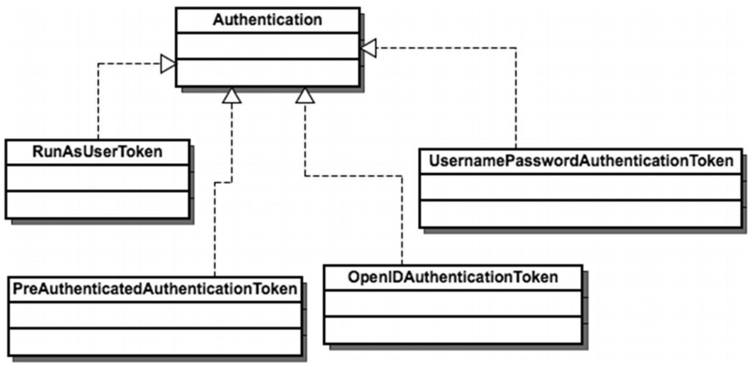

Authentication
RunAsUserToken
UsernamePasswordAuthenticationToken
PreAuthenticatedAuthenticationToken
OpenIDAuthenticationToken

Figure 4-13 Authentication hierarchy

An `Authentication` object is used 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 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 is created, storing your username, password, and permissions, which are technically known as principal, credentials, and authorities, respectively.

`Authentication` is a simple interface, as Listing 4-3 shows.

Note There are many implementations of the `Authentication` interface. In this book, we typically refer to the general `Authentication` interface when we are not interested in the implementation type. Of course, we refer to the concrete classes when discussing the specifics of an implementation detail.

```
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;

}
```

Listing 4-3
The Authentication Interface

As Figure 4-13 shows, currently, there are a few implementations of `Authentication` in the framework.

- `UsernamePasswordAuthenticationToken`: This simple implementation contains the username and password information of the authenticated (or pending authentication) user. It is the most common `Authentication` implementation used throughout the system, as many `AuthenticationProvider` objects depend directly on this class.
- `PreAuthenticatedAuthenticationToken`: This implementation exists for handling preauthenticated `Authentication` objects. Preauthenticated authentications are those where an external system handles the actual authentication process. Spring Security only extracts the principal (or user) information from the external system's messages.
- `RunAsUserToken`: This implementation is used by the `RunAsManager`, which the Security Interceptor calls 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

At the heart of Spring Security's authentication model is the SecurityContextHolder. It contains the SecurityContext.

The SecurityContextHolder is where Spring Security stores the details of who is authenticated. Spring Security does not care how the SecurityContextHolder is populated. It is used as the currently authenticated user if it contains a value.

Here is an example of SecurityContextHolder usage.

```
SecurityContext context = SecurityContextHolder.createEmptyContext();
```

```
Authentication authentication =
    new TestingAuthenticationToken("username", "password", "ROLE_USER");
context.setAuthentication(authentication);

SecurityContextHolder.setContext(context);
```

The interface `org.springframework.security.core.context.SecurityContext` (its implementation is SecurityContextImpl) is 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 to handle this `SecurityContext`—for example, one `SecurityContext` per thread (default), one global `SecurityContext`, or a custom strategy. The class diagram for these classes can be seen in Figure 4-14, and Listings 4-4 and 4-5 show the two classes.

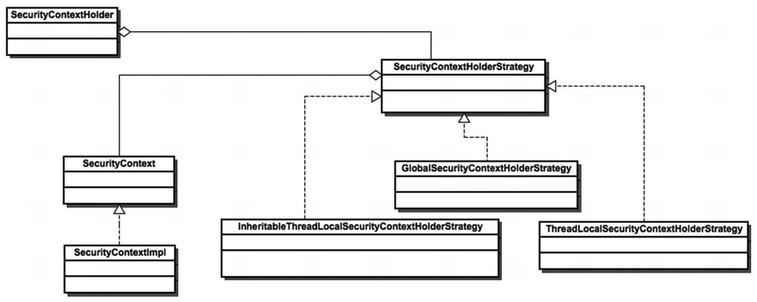

SecurityContextHolder
SecurityContextHolderStrategy
SecurityContext
GlobalSecurityContextHolderStrategy
SecurityContextImpl
InheritableThreadLocalSecurityContextHolderStrategy
ThreadLocalSecurityContextHolderStrategy

Figure 4-14 SecurityContext and SecurityContextHolder

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

import org.springframework.security.core.Authentication;

import java.io.Serializable;

public interface SecurityContext extends Serializable {

    Authentication getAuthentication();
void setAuthentication(Authentication authentication);

}
```

Listing 4-4
SecurityContext Interface

The entire `SecurityContextHolder` reference is on GitHub at **https://github.com/spring-projects/spring-security/blob/master/core/src/main/java/org/springframework/security/core/context/SecurityContext**

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

import org.springframework.util.ReflectionUtils;

import java.lang.reflect.Constructor;

public class SecurityContextHolder {

public static final String MODE_THREADLOCAL = "MODE_THREADLOCAL";

public static final String

  MODE_INHERITABLETHREADLOCAL = "MODE_INHERITABLETHREADLOCAL";

public static final String MODE_GLOBAL = "MODE_GLOBAL";

public static final String SYSTEM_PROPERTY = "spring.security.strategy";

private static String strategyName = System.getProperty(SYSTEM_PROPERTY);

private static SecurityContextHolderStrategy strategy;

private static int initializeCount = 0;

static {
```

```
initialize();

    }

public static void clearContext() {

strategy.clearContext();

    }

public static SecurityContext getContext() {

return strategy.getContext();

    }

public static int getInitializeCount() {

return initializeCount;

    }

private static void initialize() {

if ((strategyName == null) || "".equals(strategyName)) {

strategyName = MODE_THREADLOCAL;

        }

if (strategyName.equals(MODE_THREADLOCAL)) {

strategy = new ThreadLocalSecurityContextHolderStrategy();

        } else if (strategyName.equals(MODE_INHERITABLETHREADLOCAL)) {

strategy = new InheritableThreadLocalSecurityContextHolderStrategy();

        } else if (strategyName.equals(MODE_GLOBAL)) {
```

```
strategy = new GlobalSecurityContextHolderStrategy();

        } else {

try {

                Class<?> clazz = Class.forName(strategyName);

                Constructor<?> customStrategy = clazz.getConstructor();

strategy = (SecurityContextHolderStrategy) customStrategy.newInstance();

            } catch (Exception ex) {

ReflectionUtils.handleReflectionException(ex);

            }

        }

initializeCount++;

    }

public static void setContext(SecurityContext context) {

strategy.setContext(context);

    }

public static void setStrategyName(String strategyName) {

SecurityContextHolder.strategyName = strategyName;

initialize();

    }

public static SecurityContextHolderStrategy getContextHolderStrategy() {
```

```
return strategy;

    }

public static SecurityContext createEmptyContext() {

return strategy.createEmptyContext();

    }

public String toString() {

        return "SecurityContextHolder[strategy='" + strategyName + "'; initializeCount=" + initializeCo

    }

}
```

Listing 4-5
SecurityContextHolder Class

AuthenticationProvider

`AuthenticationProvider` is the main entry point for authenticating an `Authentication` object. This interface has only two methods, as Listing **4-6** shows. This is one of the major extension points of the framework, as you can tell by the many classes that currently extend this interface. Each implementing class deals with a particular external provider to authenticate against. So, if you come across a particular provider that is not supported and need to authenticate against it, you probably need to implement this interface with the required functionality. You see examples of this later in the book.

AuthenticationProvider (see Figure **4-15**) is very similar to AuthenticationManager, but it has an extra method that can be used to call a query if it supports a given Authentication type, as shown here.

```
public interface AuthenticationProvider {
        Authentication authenticate(Authentication authentication)
                        throws AuthenticationException;
        boolean supports(Class<?> authentication);
}
```

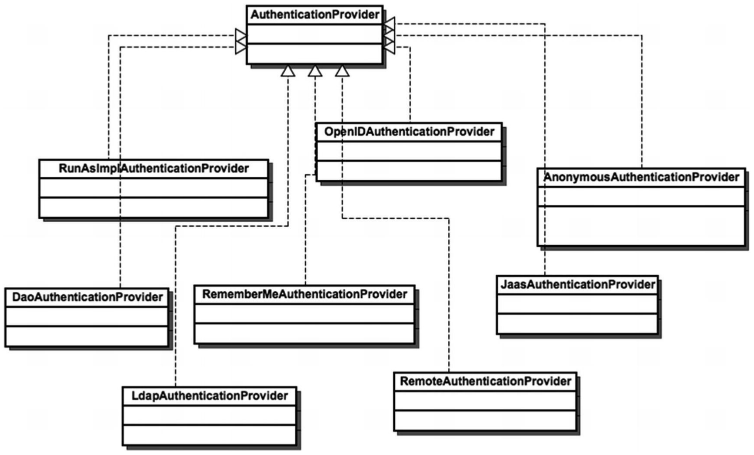

AuthenticationProvider
OpenIDAuthenticationProvider
RunAsImplAuthenticationProvider
AnonymousAuthenticationProvider
DaoAuthenticationProvider
RememberMeAuthenticationProvider
JaasAuthenticationProvider
LdapAuthenticationProvider
RemoteAuthenticationProvider

Figure 4-15 AuthenticationProvider hierarchy

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

```
CasAuthenticationProvider
JaasAuthenticationProvider
DaoAuthenticationProvider
RememberMeAuthenticationProvider
LdapAuthenticationProvider
```

The entire `AuthenticationProvider` reference is on GitHub at
https://github.com/spring-projects/spring-

security/blob/master/core/src/main/java/org/springframework/security/authentication/Authenticatio

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

import org.springframework.security.core.Authentication;

import org.springframework.security.core.AuthenticationException;

public interface AuthenticationProvider {

    Authentication authenticate(Authentication authentication) throws AuthenticationException;

boolean supports(Class<?> authentication);

}
```

Listing 4-6
AuthenticationProvider Interface

AccessDecisionManager

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

The `AccessDecisionManager` interface can be seen in Listing **4-7**. Its UML class diagram is shown in Figure **4-16**.

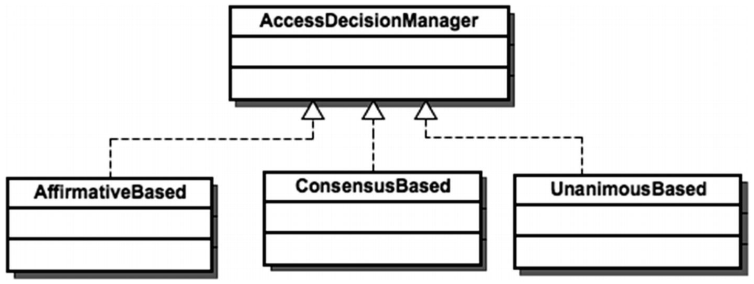


Figure 4-16 AccessDecisionManager hierarchy

The entire `AccessDecisionManager` reference is on GitHub at https://github.com/spring-projects/spring-security/blob/master/core/src/main/java/org/springframework/security/access/AccessDecisionManager

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

import java.util.Collection;

import org.springframework.security.authentication.InsufficientAuthenticationException;

import org.springframework.security.core.Authentication;

public interface AccessDecisionManager {

void decide(Authentication authentication, Object object, Collection<ConfigAttribute> configAttributes)

throws AccessDeniedException, InsufficientAuthenticationException;

boolean supports(ConfigAttribute attribute);

boolean supports(Class<?> clazz);

}
```

Listing 4-7 AccessDecisionManager

The current `AccessDecisionManager` implementations delegate to voters, but they work slightly differently. The current voters, described in the following sections, are defined in the package `org.springframework.security.access.vote`.

More information about AccessDecisionManager is at **https://docs.spring.io/spring-security/reference/servlet/authorization/architecture.html#authz-access-decision-manager**.

AffirmativeBased

This access decision manager calls all its configured voters, and if any of them vote that access should be granted, this is enough for the access decision manager to allow access to the secured resource. If no voters grant access or at least one votes 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 whether access should be granted or not when all voters abstain.

ConsensusBased

This access decision manager implementation calls all its configured voters to decide 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 more voters grant access than voters deny 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. This variable's value is true by default, and access is granted. When all voters abstain, the access decision is decided the same way as it is for the `AffirmativeBased` manager.

UnanimousBased

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

AccessDecisionVoter

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

The `org.springframework.security.access.AccessDecisionVoter` interface is very simple as shown in Listing **4-8**.

The main method is `vote`, and as can be deduced from the interface, it 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 means that the `Authentication`'s authorities are compared against the resource's security attributes looking for matches.

The following are the current `AccessDecisionVoter` implementations.

- `org.springframework.security.access.annotation.Jsr250Voter` votes on resources secured with `JSR 250` annotations—namely, `DenyAll`, `PermitAll`, and `RolesAllowed`. Their names are very descriptive. `DenyAll` won't allow access to the resource, independent of the security information carried by the `Authentication` object trying to access it. `PermitAll` allow access to everyone, regardless of what roles they have. The `RolesAllowed` annotation can be configured with a series of roles. If an `Authentication` object tries to access the resource, it must have one of the roles configured in the `RolesAllowed` annotation to get access granted by this voter.
- `org.springframework.security.access.prepost.PreInvocationAuthorizationAdviceVoter` votes on resources with expression configurations based on `@PreFilter` and `@PreAuthorize` annotations. `@PreFilter` and `@PreAuthorize` annotations support a `value` attribute with a SpEL expression. The `PreInvocationAuthorizationAdviceVoter` is in charge of evaluating the SpEL expressions (of course with the help of Spring's SpEL evaluation mechanism) provided in these annotations. SpEL expressions are discussed throughout this book so this concept becomes clearer as the book advances.
- `org.springframework.security.access.vote.AbstractAclVoter`: This is the abstract class with the skeleton to write voters dealing with domain ACL rules so that other implementing class builds on its functionality to add voting behavior. Currently, it is implemented in `AclEntryVoter`, which votes on users' permissions on domain objects. This voter is covered in Chapter 7.
- `org.springframework.security.access.vote.AuthenticatedVoter` votes whenever a `ConfigAttribute referencing any` of the three possible levels of authentication is present on the secured resource. The three levels are `IS_AUTHENTICATED_FULLY`, `IS_AUTHENTICATED_REMEMBERED`, and `IS_AUTHENTICATED_ANONYMOUSLY`. The voter emits a positive vote if the `Authentication` object's authentication level matches (or is a stronger level in the hierarchy `IS_AUTHENTICATED_FULLY > IS_AUTHENTICATED_REMEMBERED > IS_AUTHENTICATED_ANONYMOUSLY`) the authentication level configured in the resource.
- `org.springframework.security.access.vote.RoleVoter` is perhaps the most commonly used voter of them all. This voter, by default, can vote on resources that have `ConfigAttribute`(s) containing security metadata starting with the prefix ROLE_ (which can be overrid-

den). When an `Authentication` object tries to access the resource, its `GrantedAuthorities are` 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 use this voter and support SpEL expressions in web security, `the use-expressions="true"` attribute must be added to the `<http>` element.

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

The entire `AccessDecisionVoter` reference is on GitHub at https://github.com/spring-projects/spring-security/blob/master/core/src/main/java/org/springframework/security/access/AccessDecisionVoter.j

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

import java.util.Collection;

import org.springframework.security.core.Authentication;

public interface AccessDecisionVoter<S> {

int ACCESS_GRANTED = 1;

int ACCESS_ABSTAIN = 0;

int ACCESS_DENIED = -1;

boolean supports(ConfigAttribute attribute);

boolean supports(Class<?> clazz);

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

}
```

Listing 4-8
AccessDecisionVoter Interface

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` uses the provided user name to look up the rest of the required user data from the datastore. It defines just one method, as shown in Listing 4-9. The hierarchy is shown in Figure 4-17.

```
public interface UserDetailsService {

  UserDetails loadUserByUsername(String username) throws  UsernameNotFoundException;

}
```

Listing 4-9
UserDetailsServicepackage org.springframework.security.core.userdetails

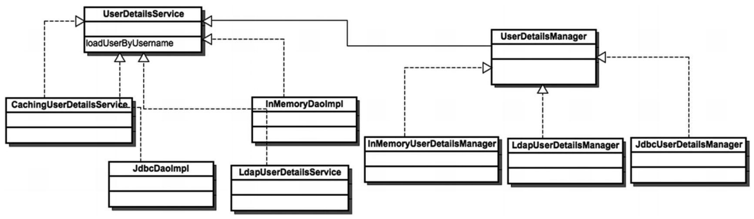


Figure 4-17 `UserDetailsService` hierarchy

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

Listing 4-10 shows the `AuthenticationUserDetailsService` interface. The two main strategies (AuthenticationUserDetailsService and UserDetailsService) are used to retrieve user information when attempting authentication. They are usually called from the particular `AuthenticationProvider` used in the application. For example, the `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;

}
```

Listing 4-10
AuthenticationUserDetailsService

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 can also be accessed later in the system from any point with access to `SecurityContext`. Normally, developers implement this interface to store user details they need or want (like email, telephone, address, and so on). Later, they can access this information, which is 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 `org.springframework.security.core.userdetails.User` class, available in the framework's core, as the `UserDetails` implementation returned by the method `loadUserByUsername`. However, this is another of those configurable points of the framework, and you can easily create your own `UserDetails` `implementation` and use that in your application. Listing **4-11** shows the `UserDetails` interface.

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

import org.springframework.security.core.Authentication;

import org.springframework.security.core.GrantedAuthority;

import java.io.Serializable;

import java.util.Collection;

public interface UserDetails extends Serializable {

Collection<? extends GrantedAuthority> getAuthorities();

String getPassword();

String getUsername();

boolean isAccountNonExpired();

boolean isAccountNonLocked();

boolean isCredentialsNonExpired();

boolean isEnabled();

}
```

Listing 4-11
UserDetails Interface

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 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 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 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 data definition language (DDL) for configuring the database comes with the framework and is in the ACL module.

ACL security is covered in Chapter 7.

JSP Taglib

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

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

Good Design and Patterns in Spring Security

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

The code for Spring Security is publicly available on GitHub at https://github.com/spring-projects/spring-security.

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

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

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

As you might see from the description of the main components of the framework, Spring Security itself is built with good design principles and patterns in mind. Let's briefly look at some of the things we find interesting in the framework, and from which you can learn.

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

Strategy Pattern

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

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

Decorator Pattern

Built into Spring's core AOP framework, you can find the Decorator pattern—mostly in the way that your annotated business classes and methods get security constraints applied to them. Basically, your objects have only certain meta information related to the security constraints that should be applied to them, and then by some "Spring magic" they get wrapped with security handling. Listing 4-12 shows the invoke method of `MethodSecurityInterceptor`. You can see how the objects are decorated with prefunctionality and postfunctionality surrounding the actual method's invocation.

```
public Object invoke(MethodInvocation mi) throws Throwable {

InterceptorStatusToken token = super.beforeInvocation(mi);
```

```
        Object result = mi.proceed();


returnsuper.afterInvocation(token, result);


    }
```

Listing 4-12

MethodSecurityInterceptor’s Invoke Method

SRP

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

DI

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

Summary

This was a complex chapter, but going through the inner workings of a software tool is the best way to understand it and take advantage of it. And that is what you did. You got an in-depth explanation of Spring Security’s architecture, major components, and how it works from the inside.

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

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

You looked at some code snippets from the framework itself to better appreciate its work and better understand why things work the way they do.

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

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

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