12

Spring Security

Jorge Simão, Ph.D.

Content

Spring Security Overview

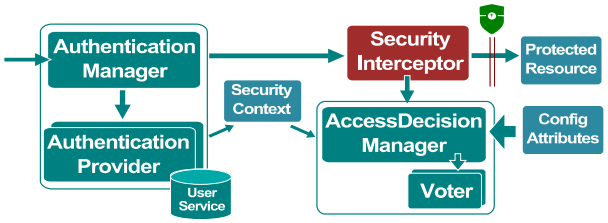
Spring Security is a security framework for Java applications, based on the **Spring Framework**. **Spring Security** is very flexible supporting a wide range of authentication protocols (e.g Form, Basic, OpenID), and several access-control mechanisms – with security policies defined at the level of the URL (in a web environment), at the level of the Java method, or ACLs for fine-grained resource protection. Access rules can be expressed also in several ways including simple role and group based-access, and SPEL scripts. **Spring Security** provides mostly a self-contained security solution, making applications and configuration fully portable across JEE application servers, Servlet containers, and other deployment environments.

Spring Security supports the same configuration styles as **Spring Framework**, including Java configuration classes, *annotation-driven* configuration, and XML configuration with namespace support. Dependencies for **Spring Security** are organized in several modules, including: core abstractions, web-specific, and XML configuration. Modules for additional authentication models are also provided (e.g. LDAP, CAS, OpenID, Kerberos, OAuth2, etc).

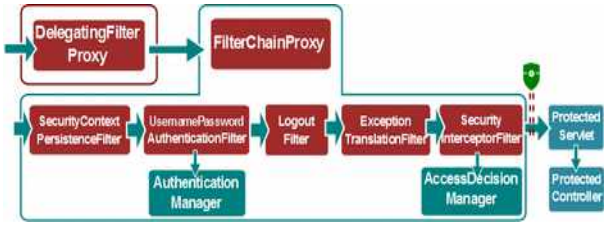
Spring Security Architecture

Spring Security architecture consists of two main sub-systems – *authentication* and *authorization* (aka. *access-control*). The authentication sub-system is responsible to establish the validity of the client (*principal*) *credentials* (e.g. username/password, or access-token). An **AuthenticationManager** component is responsible to coordinate one or more **AuthenticationProvider**, each implementing one authentication approach. Access-control is coordinated by an **AccessDecisionManager** component. The information defining the the identity and authorities of a principal are stored in a contextualized object **SecurityContext**, which works an integration data-structure for the two main sub-systems.

Enforcing a security policy always involves an act of pre-emption – performed by an *interceptor* – to check if access is allowed. To implement URL based security policies in a web environment a **Filter** based solution is used to perform the interception. For method level access-control a Spring AOP *proxy* and *around-advice* is used. Figure below depicts a generic representation of **Spring Security** architecture.



In a web environment, **Spring Security** uses a pipeline of collaborating Filters each responsible for part of the overall security solution. To simplify configuration steps, a "master" **Filter** of class **FilterChainProxy** is responsible to assemble the pipeline. One of the created Filters – **FilterSecurityInterceptor**, is responsible to preempt threads and interact with the **AccessDecisionManager**. Filter **UsernamePasswordAuthenticationFilter**, used in Form-based authentication, is responsible to authenticate the principal by extracting the credentials from a request and by interacting with the **AuthenticationManager**. Figure below depicts the **Spring Security** Filter pipeline in a web environment.



Spring Security Configuration

Spring Security can be configured both with Java and XML. In Servlet 3.0+ containers, is possible to do all the configuration in Java (i.e. no need for *web.xml* file). The annotation **@EnableWebSecurity** is used in a Java

@Configuration class to bootstrap the configuration of **Spring Security** in a web environment. This creates the **FilterChainProxy** filter as a Spring managed component (bean) named **springSecurityCheck**, which in turn creates the security filter pipeline.

Customization of Spring Security can be done by making the configuration class annotated with **@EnableWebSecurity** implement the interface **WebSecurityConfigurer**, or more conveniently extend class **WebSecurityConfigurerAdapter**, and overrides the callback methods. Method **configureGlobal()** is used to configure global elements, such as one or more **AuthenticationProvider** and **UserDetailsService**. Method **configure()** is use to configure the HTTP specific elements, such as the selection of the authentication protocol to use and URL based access-control rules.

Similar configuration can be done in XML using the **<security:*>** namespace. Element **<http>** defines the HTTP-specific configuration. Element **<authentication-manager>** encapsulates the configuration for user authentication.

» Example: Configure Spring Security for WebApps [Java]

```
@EnableWebSecurity
@Configuration
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

  @Autowired
  public void configureGlobal(AuthenticationManagerBuilder
auth) throws Exception {
    auth.inMemoryAuthentication()
      .withUser("user")
        .password("userpa$$").roles("USER").and()
      .withUser("admin")
        .password("adminpa$$").roles("USER", "ADMIN");
  }

  @Autowired
  public void configure(HttpSecurity http) throws Exception {
    http.authorizeRequests()
      .anyRequest().authenticated().and()
      .formLogin();
  }
}
```

» Example: Configure Spring Security for WebApps [XML]

```
<http>
  <intercept-url pattern="/static/**" access="permitAll"/>
  <intercept-url pattern="/Login" access="permitAll"/>
  <intercept-url pattern="/" access="permitAll"/>
  <intercept-url pattern="/**" access="authenticated"/>
  <form-login />
</http>
```

To integrate the **FilterChainProxy** in a Servlet container, a container managed filter of type **DelegatingFilterProxy** name should be configured with name **springSecurityCheck**. In a Servlet 3.0+ environment, this can be done by defining a class extending **AbstractSecurityWebApplicationInitializer**. Alternatively, the **DelegatingFilterProxy** can be defined explicitly in the *web.xml* application descriptor file.

» Example: Configure DelegatingFilterProxy [Java]

```
public class SecurityWebApplicationInitializer
extends AbstractSecurityWebApplicationInitializer {}
```

» Example: Configure DelegatingFilterProxy [XML]

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.sf.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Table below summarizes **Spring Security** annotations. Some of these annotations will be covered in following sections.

Parameter	Description
`@EnableWebSecurity`	Enable Java-based configuration
`@EnableGlobalMethodSecurity`	Enable method-level security
`@EnableGlobalAuthentication`	Enable Java-based configuration of `AuthenticationManager`
`@Secured`	Define role-based access-control for method
`@RolesAllowed`	Define role-based access-control for method [JSR-250 annotation]
`@PreAuthorize`	Define role-based access-control for method with SPEL expression
`@PostAuthorize`	Define role-based access-control for method with SPEL [check done on return]

Form-Based Autentication

Form-based authentication involves using a HTML form to submit *username–password* credentials as *form parameters*. Method **formLogin()**, in **HttpSecurity**, is used to enable **FORM** authentication. Equivalently, XML element **<form-login>** can be used for the same purpose.

By default, **Spring Security** auto-generates an HTML *login form*. To customize the login form, the method **loginPage()** can be used when setting up **HttpSecurity**. When done with XML configuration, the attribute **loginPage** should be used. When providing custom HTML login forms, the **action** (URL) attribute should match the setting specified in **formLogin()** or **<form-login>**, or **/login** for the default setting. Additionally, the form parameters names **username** and **password** should be used for the credentials input fields (as default).

» Example: Configuring the Login Form Page

```
public void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests()
    .anyRequest().authenticated().and()
    .formLogin().loginPage("/login").permitAll();
}
```

» Example: A Custom HTML Login Form [JSP]

```
<c:url value="/login" var="loginUrl"/>
<form action="${loginUrl}" method="post">
  <c:if test="${not empty param.error}"><p>Try Again</p></c:if>
  <p><label>Username</label> <input name="username"/></p>
  <p><label>Password</label> <input type="password"
name="password"/></p>
  <button type="submit">LOGIN</button>
</form>
```

When a user completes an interaction session, it is desirable *logout* the user to clear the session state, including the **SecurityContext**. This is done by configuring a **LogoutFilter**, that handler HTTP POST request to a logout URL – **/logout**, by default. This is configured in Java with method **logout()**, or in XML with element **<logout />**. After logout the user is redirect to a logout URL – the home page **/**, by default.

» Example: Configuring Logout Filter [Java]

```
protected void configure(HttpSecurity http) throws Exception {
  http.authorizeRequests(). … .and()
    .formLogin().loginPage("/login").permitAll().and()
    .logout()
     .logoutUrl("/logout")
     .logoutSuccessUrl("/thankyou");
}
```

» Example: Configuring Logout Filter [XML]

```
<http>
  <form-login />
  <logout logout-success-url="/thankyou"/>
</http>
```

Web/URL Based Access-Control

An URL-based *access-control policy* is defined using the fluent API made available by calling **authorizeRequests()** in **HttpSecurity**. Each *access-rule* defines a pattern to match in the request URL, and optionally the HTTP method, followed by required authorities. Method **antMatchers()** is used to defined rules using Ant-like expressions. Regular expressions can also be used with method **regexMatchers()**.

Similar configuration can be done in XML by using element **<intercept-url>**, inside **<http>**, to define an access-rule. By default, SPEL script expression should be used to express the required access-policy and authorities. Alternatively, role-based access rules can be defined, by setting attribute **use-expresions="false"** in element **<http>**. Access rules, both in Java and XML, are matched in order. So the more specific rules should come earlier, followed by the more general rules.

» Example: Defining Access-Control Policy Rules [Java]

```
public void configure(HttpSecurity http) throws ... {
  http.authorizeRequests()
    .antMatchers("/static/**", "/login", "/").permitAll()
    .antMatchers("/admin/**").hasRole("ADMIN")
    .anyRequest().authenticated().and()
    .formLogin().loginPage("/login").permitAll();
}
```

» Example: Defining Access-Control Rules w/ SPEL [XML]

```
<http use-expressions="true" pattern="**">
  <intercept-url pattern="/static/**" access="permitAll"/>
  <intercept-url pattern="/login" access="permitAll"/>
  <intercept-url pattern="/" access="permitAll"/>
  <intercept-url pattern="/admin/**"
access="hasRole('ADMIN')"/>
  <intercept-url pattern="/**" access="authenticated"/>
  <form-login />
</http>
```

» Example: Role-Based Access-Control Rules [XML]

```
<http use-expressions="false" pattern="/api/**">
  <intercept-url pattern="/admin/**" access="ROLE_ADMIN"/>
  <intercept-url pattern="/**" method="POST"
access="AUTHENTICATED"/>
  <form-login />
</http>
```

User Databases

The descriptor of type **AuthenticationManagerBuilder** in method **configureGlobal()** provides a fluent API to configure the **AuthenticationManager**. An **AutenticationProvider** can be defined to use a service or repository of user credentials and authorities of type **UserDetailsService**. The credentials returned by the service are compared with the ones send by the user over the wire to be validated and perform the authentication.

Method **inMemoryAuthentication()** creates an **AutenticationProvider** with an in-memory repository of users. This is suitable to define ad-hoc (made up) users for development purposes. For production environments a database is more suitable. Method **jdbcAuthentication()** creates an **AutenticationProvider** which uses user repository for relational database and configures it with a **JDBC DataSource**.

As a security precaution, user passwords should be stored encrypted in the database. The encryption strategy used when the password was stored (on user registration or password change) should be applied also to the password received during authentication. The method **passwordEncoder()** allows to specify the encryption strategy to use via a **Password Encoder**. The strategy **StandardPasswordEncoder** uses **SHA-256** algorithm for encryption. It also supports password

padding with an additional “entropy” bytes (*salt*).

» Example: Defining an InMemory UserDetailsService

```
@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception {
  auth.inMemoryAuthentication()
    .withUser("user").password("userpa$$").roles("USER").and()
    .withUser("admin").password("adminpa$$").roles("USER",
"ADMIN")
    .and().passwordEncoder(
       new StandardPasswordEncoder("$ecret-key99$"));
}
```

» Example: Defining a JDBC UserDetailsService

```
@Autowired
DataSource dataSource;

@Autowired
public void configureGlobal(AuthenticationManagerBuilder auth)
throws Exception {
  auth.jdbcAuthentication().dataSource(dataSource)
}
```

The JDBC **UserDetailsService** assumes a database schema with tables USERS and AUTHORITIES. Table below show the default SQL queries executed. This queries can also be changed with setters in the fluent API. Policies based on group membership and group authorities are also supported.

Query Purpose & Setter	SQL Query [default]
Find user credentials by username. usersByUsernameQuery()	`select username,password,enabled from users where username = ?`
Find user authorities by username (e.g. roles). authoritiesByUsernameQuery()	`select username,authority from authorities where username = ?`
Find user's groups authorities by username (e.g. roles) groupAuthoritiesByUsername()	`select g.id, g.group_name, ga.authority from groups g, group_members gm, group_authorities ga where gm.username = ? and g.id = ga.group_id and g.id = gm.group_id`

Similar configuration of the **AuthenticationManager** and **AuthenticationProvider**, can be done in XML using elements **<authentication-manager>** and **<authentication-provider>**. Element **<user-service>** defines an **InMemoryUserDetailsService**. Element **<jdbc-user-service>** defines a **JdbcUserDetailsService**.

» Example: Defining a JDBC UserService [XML]

```
<authentication-manager>
  <authentication-provider>
    <jdbc-user-service data-source-ref="dataSource" />
    <password-encoder hash="sha-256">
      <salt-source system-wide="$super$ecret123$"/>
    </password-encoder>
  </authentication-provider>
</authentication-manager>
```

» Example: Defining an InMemory UserService [XML]

```
<authentication-provider>
  <user-service>
    <user name="user" password="us$r" authorities="ROLE_USER" />
    <user name="admin" password="$$" authorities="ROLE_ADMIN" />
  </user-service>
</authentication-provider>
```

Combined / Alternative Configuration

Applications often need to combine different configuration options, such as using different types of authentication, for different types of endpoints (e.g. using Form-based authentication for web endpoints, and BASIC authentication for REST-WS endpoints). This can be done in Java by using more than one **@Configuration** class, and using a URL prefix matcher with method **HttpSecurity.antMatcher()**. Annotation **@Order** can be used to guarantee that more specific matchers are evaluate first. In XML, same result is achieved using the attribute **pattern** in (multiple) **<http>** elements.

When alternative configurations are to be used in different environments, *environment profiles* should be used (e.g. using **InMemoryUserDetailsService** in development, and a **JdbcUserDetailsService** in production).

» Example: Combined Configuration Policies [Java]

```
@Configuration
@Order(1)
public static class ApiWebSecurityConfigurationAdapter extends
WebSecurityConfigurerAdapter {
  protected void configure(HttpSecurity http) throws ... {
    http.antMatcher("/api/**")
      .authorizeRequests(). ... .and()
      .httpBasic();
  }
}

@Configuration
public static class FormLoginWebSecurityConfigurerAdapter
extends WebSecurityConfigurerAdapter {
  @Override
  protected void configure(HttpSecurity http) throws ... {
    http.authorizeRequests(). ... .and()
      .formLogin();
    }
}
```

» Example: Combined Configuration Policies [XML]

```
<http pattern="/login" security="none" />
<http pattern="/api/**">
  <intercept-url pattern="/**" access="AUTHENTICATED"/>
  <basic-auth />
</http>
<http pattern="/**">
  <intercept-url pattern="/admin/**" access="ROLE_ADMIN"/>
  <form-login />
</http>
```

Method-Level Access-Control

Spring Security allows access-control policies to be defined at the level of the Java method. This is implementing by wrapping the Spring beans with protected methods into a *proxy* that calls a *security advice*. The advice interacts with the **AccessDecisionManager** in a way comparable to what the **SecurityInterceptorFilter** does for URL-based policies. The annotation **@EnableGlobalMethodSecurity** enables the creation of the security advices. The **@Secured** annotation is used at the method or class level to define the access-policy for intercepted methods. The **value()** attribute of **@Secured** specifies a list of roles (authorities) that the **Principal** may hold so that threads running on its behalf are allowed to invoke the method. If the security check fails a **NotAuthorizedException** is thrown. The attribute **securedEnable** of annotation **@EnableGlobalMethodSecurity** should also be set to **true**.

Spring Security also supports the standard JSR-250 annotation **@javax.annotation.RolesAllowed**, by setting attribute **js250Enabled** on **@EnableGlobalMethodSecurity** to **true**. For SPEL expressions as access-rules use annotation **@PreAuthorize** (or **@PostAuthorized**), and set attribute **prePostEnabled** of **@EnableGlobalMethodSecurity** to **true**.

» Example: Enabling Method-Level Security

```
@EnableGlobalMethodSecurity(securedEnabled = true,
jsr250Enabled=true, prePostEnabled=true)
@Configuration
public class MethodSecurityConfig {
}
```

» Example: A Secured Service Interface

```
public interface OrderService {
    @Secured("IS_AUTHENTICATED")
    public Order submit(Order Order);

    @Secured("IS_AUTHENTICATED_ANONYMOUSLY")
    public Order getOrder(Long id);
}
```

Method-level security can also be configured in XML with element **<global-method-security>**. Since method-level security is based on **Spring AOP** run-time, **AspectJ**-style *point-cut expressions* can be used to secure many methods declaratively.

» Example: Enabling Method Security [XML]

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

» Example: Securing Methods w/ Point-Cut Expressions

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

Basic & Digest Authentication

For non-interactive applications and HTTP endpoints, such as REST-WS, form-based authentication is not suitable. In these scenarios, requests should include the user credentials in a request header – often the HTTP **Authorization** header. **Spring Security** has support for several protocols that follow this approach, such as HTTP **BASIC** and **DIGEST** authentication.

BASIC authentication is a simple protocol, specified in RFC-1945, that sends credentials unencrypted in format ***Encode64[username:password]***. **BASIC** authentication is particularly simple to configure in **Spring Security**, both in Java and XML configuration, since there is fluent API and XML namespace support. Method **httpBasic()** configures a **Filter** that extract the credentials from the **Authorization** header and perform the authentication. XML element **<http-basic>** has the same effect. Note that **BASIC** authentication is considered unsecure, like **FORM** authentication, unless the credentials are sent trough a secured channel (i.e. encrypted with a **SSL/TLS** connection).

DIGEST authentication, specified in RFC-2617 and RFC-2069, fixes some of the vulnerabilities of **BASIC** authentication by sending credentials encrypted (with MD5 algorithm). Configuration in **Spring Security** is more explicit since there is no fluent API or XML namespace support. A **Filter** of type **DigestAuthenticationFilter** should be configured to extract and validate the credentials. A bean of type **DigestAuthenticationEntryPoint** should also be configure – which is called when authentication is required to access a protected resource, to send the correct HTTP status code (**401 Unauthorized**), and set the **WWW-Authenticate** header. Note that in DIGEST authentication user credentials should be stored in plain format, or encrypted in MD5. Note also that when using a secured channel there is no need to use DIGEST authentication.

» Example: Configuring Basic Authentication [Java]

```
protected void configure(HttpSecurity http) throws Exception {
  http.antMatcher("/api/**")
    .authorizeRequests(). … .and()
    .httpBasic();
}
```

» Example: Configuring Basic Authentication [XML]

```
<http use-expressions="false" pattern="/api/**">
  <intercept-url pattern="/admin/**" access="ROLE_ADMIN"/>
  <intercept-url pattern="**"
access="IS_AUTHENTICATED_ANONYMOUSLY"/>
  <http-basic />
</http>
```

» Example: Configuring Digest Authentication [XML]

```
<bean id="digestFilter"
class="org.sf.sec.web.auth.www.DigestAuthenticationFilter"
  p:userDetailsService-ref="jdbcDaoImpl"
```

```
  p:authenticationEntryPoint-ref="digestEntryPoint"
  p:passwordAlreadyEncoded="true" p:userCache-ref="userCache"/>

<bean id="digestEntryPoint"
class="org.sf.sec.web.auth.www.DigestAuthenticationEntryPoint"
  p:realmName="MyApp" p:key="$erver$secret#XYZ"
  p:nonceValiditySeconds="10"/>
```

Secure Encrypted Channels

To be secure, **FORM** and **BASIC** authentication require sending credentials trough a secured encrypted **SSL/TLS** connection. Secured connections can also be used to encrypt payload data and increased privacy. **Spring Security** can be configured to force the use of a secured channel, by setting attribute **requires-channel="https"** in element **<intercept-url>**. Although this can usually also be done via direct configuration of the *Servlet* container, doing it with **Spring Security** increases the portability of the application. Non-default port-mappings can be defined with element **<port-mapping>**.

» Example: Forcing HTTPS Secured Connection on Login

```
<http use-expressions="true" pattern="**" >
  <intercept-url pattern="/login" access="permitAll" requires-
channel="https"/>
  <intercept-url pattern="/static/**" access="permitAll"/>
  <intercept-url pattern="/" access="permitAll"/>
  <intercept-url pattern="/**" access="authenticated"/>
  <form-login />
  <port-mappings>
    <port-mapping http="8080" https="9443"/>
  </port-mappings>
</http>
```

RememberMe (Cookie) Authentication

Remember-me (Cookie) Authentication allow users to be automatically logged-in long after a session is closed by the server. **Spring Security** support two strategies for implementing this kind of authentication approach. In both approaches, a token is generated and shared with the *user-agent* (browser) as a cookie after login. In the simplest approach, the token is created by appending user info and an encrypted digest. This is configured in XML with element **<remember-me>** and setting the attribute **key** to a secret value – only know by the server, that makes the token unforgeable by malevolences third-parties (hackers). When a browser present the cookie with a valid token, the user is authenticated.

The more sophisticated approach to remember-me authentication uses a database to keep track of issued tokens. Each token is valid for a single request, at which point a new one is generated and the previous one removed. Each token also belongs to a token-series, whose value is also included in the cookie. This allows **Spring Security** to detect when a token was stolen, and make a complete token-series to be invalidated and a new one started. This approach is configured in XML by setting attribute **data-source-ref**.

» Example: Config Hash-Based Remember-me Auth

```
<http>
  <intercept-url pattern="/login" access="permitAll"/>
  <intercept-url pattern="**" access="authenticated"/>
  <form-login />
  <remember-me key="$erver$super$secret$$"/>
</http>
```

» Example: Config Persistente Remember-me Auth

```
<http>
  ...
  <form-login />
  <remember-me data-source-ref="dataSource"/>
</http>
```

ACL – Access-Control Lists

For use-cases where fine-grained access control to individual objects is required, **Spring Security** provides an out-of-the-box API and services for *Access-Control Lists* (ACL) based authorization (e.g. to authorize access based on resource ownership). An **AclService** bean provides an API to retrieve, create, update, and manage **Acl**. The default implementation **JdbcMutableAclService** uses a **DataSource** with a schema with four tables to describe ACL entrie – **ACL_SID**, **ACL_CLASS**, **ACL_OBJECT_IDENTITY**, **ACL_ENTRY**. Each entry contained in an **Acl** defines which permissions (modelled as an array of bits) a user (modelled a **PrincipalSid**) can perform on an application object (modelled as as **ObjectIndentity**). The method **Acl.isGranted()** is use to check for permissions.

» Example: Checking ACL for Object Read Access

```
@Autowired
MutableAclService aclService;
```

```
@GetMapping("/order/{id}")
public String getOrder(@PathVariable("id") Long id, Model
model, Principal principal) {
  ObjectIdentity oi = new ObjectIdentityImpl(Order.class, id);
  Sid sid = new PrincipalSid(principal.getName());
  Permission p = BasePermission.READ;
  try {
    MutableAcl acl = (MutableAcl) aclService.readAclById(oi);
    if (!acl.isGranted(singletonList(p), singletonList(sid),
false)) {
      throw new AccessDeniedException("Not in ACL!");
    }
  } catch (NotFoundException nfe) {}
  Order order = orderService.getOrder(id);
  model.addAttribute("order", order);
  return "order/show";
}
```


Cross-Site Request Forgery

Cross-Site Request Forgery (CSRF) is a category of attacks where a malicious website, disguising its intent, forwards the user to another genuine website – where the user might have been recently logged in – with a request to update data (e.g. via an HTTP POST, PUT, DELETE). In **Spring Security,** this can be prevented by including a HTTP request parameter or cookie that the attacker can not guess. By default, **Spring Security** generates the CSRF token automatically and saves it under *request-scoped* parameter **_csrf**. HTML requests making updates – such as a login credentials submission – should include this as an hidden parameter. When using the **<form:from>** tag, provided in the JSP Form **taglib** of **Spring**, this is done automatically. Disabling or fine configuration of CSRF can be done in Java with method **csrf()** or in XML with element **<csrf>**.

» Example: Adding CSRF Token as Hidden Form Param

```
<c:url value="/login" var="loginUrl"/>
<form action="${loginUrl}" method="post">
  ...
  <input type="hidden" name="${_csrf.parameterName}" value="$
{_csrf.token}"/>
  <button type="submit">LOGIN</button>
</form>
```

Extensions to Spring Security

Spring Security provides additional modules and extensions that can be for authentication and authorization, such as: **LDAP** external authentication, OpendID based-authentication, **Servlet** container Integration, **Kerberos**, **OAuth2**. Developers can also create new extensions by providing alternative implementation to the framework interfaces defined as **Spring** beans. Spring Social extensions allow users to login in application using credentials from a variety of social web-sites (e.g. Google, Facebook, Twitter).

Custom filters can also be added to the filter pipeline setup by default by Spring Method. This is one with method **HttpSecurity.addFilter()** or XML element **<custom-filter>**.

Resources

- Spning Security Referenc Manual – http://docs.spring.io/spring-security/site/docs/4.2.0.BUILD-SNAPSHOT/reference/htmlsingle/
- Spring Security Project Page – http://projects.spring.io/spring-security/

About the Author

Jorge Simão is a software engineer and IT Trainer, with two decades long experience on education delivery both in academia and industry. Expert in a wide range of computing topics, he his an author, trainer, and director (Education & Consulting) at **EInnovator**. He holds a B.Sc., M.Sc., and Ph.D. in Computer Science and Engineering.

Core Spring & Spring Security Training

Core Spring is **Pivotal**'s official four-day flagship **Spring Framework** training. In this course, students build a Spring-powered Java application that demonstrates the **Spring Framework** and other Spring technologies like Spring AOP and Spring Security in an intensely productive, hands-on setting. Completion of this training prepares participants to take a certification exam and become a **Spring Certified Professional**.
Book now an on-site training for date&location of your choice: **www.einnovator.org/course/core-spring**

EInnovator – Software Engineering School

EInnovator.org offers the best Software Engineering resources and education, in partnership with the most innovative companies in the IT-world. Focusing on both foundational and cutting-edge technologies and topics, at EInnovator software engineers and data-scientists can get all the skills needed to become top-of-the-line on state-of-the-art IT professionals.

Contacts

Training – Bookings & Inquiries
training@einnovator.org

Consultancy – Partnerships & Inquiries
consulting@einnovator.org

General Info
info@einnovator.org