**Spring security**

http://docs.spring.io/spring-security/site/docs/3.2.5.RELEASE/reference/htmlsingle/#authz-arch

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

* SecurityContextHolder, to provide access to the SecurityContext.
* SecurityContext, to hold the Authentication and possibly request-specific security information.
* Authentication, to represent the principal in a Spring Security-specific manner.
* GrantedAuthority, to reflect the application-wide permissions granted to a principal.
* UserDetails, to provide the necessary information to build an Authentication object from your application’s DAOs or other source of security data.
* UserDetailsService, to create a UserDetails when passed in a String-based username (or certificate ID or the like).

Process involved in username/password authentication

1. The username and password are obtained and combined into an instance of UsernamePasswordAuthenticationToken (an instance of the Authentication interface, which we saw earlier).
2. The token is passed to an instance of AuthenticationManager for validation.
3. The AuthenticationManager returns a fully populated Authentication instance on successful authentication.
4. The security context is established by calling SecurityContextHolder.getContext().setAuthentication(...), passing in the returned authentication object.

ExceptionTranslationFilter

offers this service, with specific responsibility

* For either returning error code 403 (if the principal has been authenticated and therefore simply lacks sufficient access - as per step seven above), or
* launching an AuthenticationEntryPoint (if the principal has not been authenticated and therefore we need to go commence step three).

<bean id="exceptionTranslationFilter"

class="org.springframework.security.web.access.ExceptionTranslationFilter">

<property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>

<property name="accessDeniedHandler" ref="accessDeniedHandler"/>

</bean>

<bean id="authenticationEntryPoint"

class="org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint">

<property name="loginFormUrl" value="/login.jsp"/>

</bean>

<bean id="accessDeniedHandler"

class="org.springframework.security.web.access.AccessDeniedHandlerImpl">

<property name="errorPage" value="/accessDenied.htm"/>

</bean>

#### AuthenticationEntryPoint

The AuthenticationEntryPoint is responsible for step three in the above list. As you can imagine, each web application will have a default authentication strategy (well, this can be configured like nearly everything else in Spring Security, but let’s keep it simple for now). Each major authentication system will have its own AuthenticationEntryPoint implementation, which typically performs one of the actions described in step 3

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

#### Authentication Mechanism

Once your browser submits your authentication credentials (either as an HTTP form post or HTTP header) there needs to be something on the server that"collects" these authentication details. By now we’re at step six in the above list. In Spring Security we have a special name for the function of collecting authentication details from a user agent (usually a web browser), referring to it as the "authentication mechanism". Examples are form-base login and Basic authentication. Once the authentication details have been collected from the user agent, anAuthentication"request" object is built and then presented to the AuthenticationManager.

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

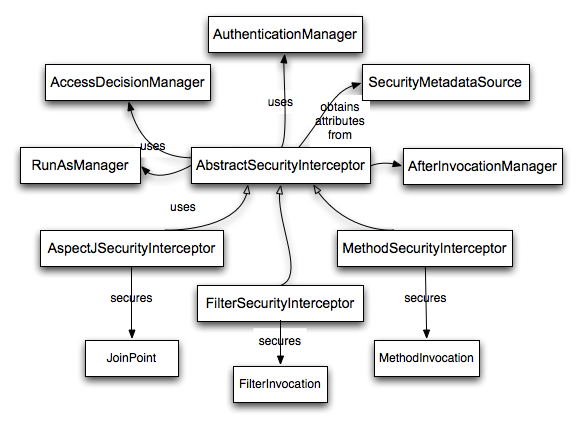
#### 1.4.4. Storing the SecurityContext between requests

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

Many other types of application (for example, a stateless RESTful web service) do not use HTTP sessions and will re-authenticate on every request. However, it is still important that the SecurityContextPersistenceFilter is included in the chain to make sure that theSecurityContextHolder is cleared after each request.

The main interface responsible for making access-control decisions in Spring Security is the AccessDecisionManager

AbstractSecurityInterceptor and its related objects are shown in [Security interceptors and the "secure object" model](http://docs.spring.io/spring-security/site/docs/3.2.5.RELEASE/reference/htmlsingle/#abstract-security-interceptor)



### 1.6. Localization

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

<bean id="messageSource"

class="org.springframework.context.support.ReloadableResourceBundleMessageSource">

<property name="basename" value="classpath:org/springframework/security/messages"/>

</bean>

## 2. Core Services

### The AuthenticationManager, ProviderManager and AuthenticationProvider

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

#### 2.1.2. DaoAuthenticationProvider

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

#### 2.2.1. In-Memory Authentication

<user-service id="userDetailsService"> <user name="jimi" password="jimispassword" authorities="ROLE\_USER, ROLE\_ADMIN" /> <user name="bob" password="bobspassword" authorities="ROLE\_USER" /> </user-service>

This also supports the use of an external properties file:

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

The properties file should contain entries in the form

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

#### 2.2.2. JdbcDaoImpl

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

### 2.3. Password Encoding

org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder" is a good choice for security. There are also compatible implementations in other common programming languages so it a good choice for interoperability too.

If you want to generate encoded passwords directly in Java for storage in your user database, then you can use the encode method on thePasswordEncoder.

# Web Application Security

## 1. The Security Filter Chain

### 1.1. DelegatingFilterProxy

<filter> <filter-name>myFilter</filter-name> <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class> </filter> <filter-mapping> <filter-name>myFilter</filter-name> <url-pattern>/\*</url-pattern> </filter-mapping>

### 1.2. FilterChainProxy

FilterChainProxy lets us add a single entry to web.xmland deal entirely with the application context file for managing our web security beans. It is wired using a`DelegatingFilterProxy`, just like in the example above, but with the filter-name set to the bean name "filterChainProxy". The filter chain is then declared in the application context with the same bean name. Here’s an example:

<bean id="filterChainProxy" class="org.springframework.security.web.FilterChainProxy">

<constructor-arg>

<list>

<sec:filter-chain pattern="/restful/\*\*" filters="

securityContextPersistenceFilterWithASCFalse,

basicAuthenticationFilter,

exceptionTranslationFilter,

filterSecurityInterceptor" />

<sec:filter-chain pattern="/\*\*" filters="

securityContextPersistenceFilterWithASCTrue,

formLoginFilter,

exceptionTranslationFilter,

filterSecurityInterceptor" />

</list>

</constructor-arg>

</bean>

### 1.3. Filter Ordering

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

* ChannelProcessingFilter, because it might need to redirect to a different protocol
* SecurityContextPersistenceFilter, so a SecurityContext can be set up in the SecurityContextHolder at the beginning of a web request, and any changes to the SecurityContext can be copied to the HttpSession when the web request ends (ready for use with the next web request)
* ConcurrentSessionFilter, because it uses the SecurityContextHolder functionality and needs to update the SessionRegistry to reflect ongoing requests from the principal
* Authentication processing mechanisms - UsernamePasswordAuthenticationFilter, CasAuthenticationFilter,BasicAuthenticationFilter etc - so that the SecurityContextHolder can be modified to contain a valid Authentication request token
* The SecurityContextHolderAwareRequestFilter, if you are using it to install a Spring Security aware HttpServletRequestWrapper into your servlet container
* The JaasApiIntegrationFilter, if a JaasAuthenticationToken is in the SecurityContextHolder this will process the FilterChainas the Subject in the JaasAuthenticationToken
* RememberMeAuthenticationFilter, so that if no earlier authentication processing mechanism updated the SecurityContextHolder, and the request presents a cookie that enables remember-me services to take place, a suitable remembered Authentication object will be put there
* AnonymousAuthenticationFilter, so that if no earlier authentication processing mechanism updated the SecurityContextHolder, an anonymous Authentication object will be put there
* ExceptionTranslationFilter, to catch any Spring Security exceptions so that either an HTTP error response can be returned or an appropriate AuthenticationEntryPoint can be launched
* FilterSecurityInterceptor, to protect web URIs and raise exceptions when access is denied

## 2. Core Security Filters

### 2.1. FilterSecurityInterceptor

<bean id="filterSecurityInterceptor" class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor"> <property name="authenticationManager" ref="authenticationManager"/> <property name="accessDecisionManager" ref="accessDecisionManager"/> <property name="securityMetadataSource"> <security:filter-security-metadata-source> <security:intercept-url pattern="/secure/super/\*\*" access="ROLE\_WE\_DONT\_HAVE"/> <security:intercept-url pattern="/secure/\*\*" access="ROLE\_SUPERVISOR,ROLE\_TELLER"/> </security:filter-security-metadata-source> </property> </bean>

ilterSecurityInterceptor is responsible for handling the security of HTTP resources. It requires a reference to anAuthenticationManager and an AccessDecisionManager. It is also supplied with configuration attributes that apply to different HTTP URL requests.

### 2.3. SecurityContextPersistenceFilter

We covered the purpose of this all-important filter in the [Technical Overview](http://docs.spring.io/spring-security/site/docs/3.2.5.RELEASE/reference/htmlsingle/#tech-intro-sec-context-persistence) chapter so you might want to re-read that section at this point. Let’s first take a look at how you would configure it for use with a FilterChainProxy. A basic configuration only requires the bean itself

<bean id="securityContextPersistenceFilter"

class="org.springframework.security.web.context.SecurityContextPersistenceFilter"/>

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

#### 2.3.1. SecurityContextRepository

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

### 2.4. UsernamePasswordAuthenticationFilter

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

### 4.1. BasicAuthenticationFilter

BasicAuthenticationFilter is responsible for processing basic authentication credentials presented in HTTP headers.

<bean id="basicAuthenticationFilter"

class="org.springframework.security.web.authentication.www.BasicAuthenticationFilter">

<property name="authenticationManager" ref="authenticationManager"/>

<property name="authenticationEntryPoint" ref="authenticationEntryPoint"/>

</bean>

<bean id="authenticationEntryPoint"

class="org.springframework.security.web.authentication.www.BasicAuthenticationEntryPoint">

<property name="realmName" value="Name Of Your Realm"/>

</bean>

### DigestAuthenticationFilter

Digest Authentication attempts to solve many of the weaknesses of Basic authentication, specifically by ensuring credentials are never sent in clear text across the wire. Many user agents support Digest Authentication, including FireFox and Internet Explorer.

<bean id="digestFilter" class=

"org.springframework.security.web.authentication.www.DigestAuthenticationFilter">

<property name="userDetailsService" ref="jdbcDaoImpl"/>

<property name="authenticationEntryPoint" ref="digestEntryPoint"/>

<property name="userCache" ref="userCache"/>

</bean>

<bean id="digestEntryPoint" class=

"org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint">

<property name="realmName" value="Contacts Realm via Digest Authentication"/>

<property name="key" value="acegi"/>

<property name="nonceValiditySeconds" value="10"/>

</bean>

## Remember-Me Authentication

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

### 5.2. Simple Hash-Based Token Approach

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

### 5.3. Persistent Token Approach

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

create table persistent\_logins (username varchar(64) not null, series varchar(64) primary key, token varchar(64) not null, last\_used timestamp not null)

## 6. Cross Site Request Forgery (CSRF)

## 9. Anonymous Authentication

<bean id="anonymousAuthFilter" class="org.springframework.security.web.authentication.AnonymousAuthenticationFilter"> <property name="key" value="foobar"/> <property name="userAttribute" value="anonymousUser,ROLE\_ANONYMOUS"/> </bean> <bean id="anonymousAuthenticationProvider" class="org.springframework.security.authentication.AnonymousAuthenticationProvider"> <property name="key" value="foobar"/> </bean>

<bean id="filterSecurityInterceptor" class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor"> <property name="authenticationManager" ref="authenticationManager"/> <property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/> <property name="securityMetadata"> <security:filter-security-metadata-source> <security:intercept-url pattern='/index.jsp' access='ROLE\_ANONYMOUS,ROLE\_USER'/> <security:intercept-url pattern='/hello.htm' access='ROLE\_ANONYMOUS,ROLE\_USER'/> <security:intercept-url pattern='/logoff.jsp' access='ROLE\_ANONYMOUS,ROLE\_USER'/> <security:intercept-url pattern='/login.jsp' access='ROLE\_ANONYMOUS,ROLE\_USER'/> <security:intercept-url pattern='/\*\*' access='ROLE\_USER'/> </security:filter-security-metadata-source>" + </property> </bean>

## 1. Authorization Architecture

### 1.4. Hierarchical Roles

he use of a role-hierarchy allows you to configure which roles (or authorities) should include others. An extended version of Spring Security’s [RoleVoter](http://docs.spring.io/spring-security/site/docs/3.2.5.RELEASE/reference/htmlsingle/" \l "authz-role-voter), RoleHierarchyVoter, is configured with a RoleHierarchy, from which it obtains all the "reachable authorities" which the user is assigned. A typical configuration might look like this:

### 3.2. Web Security Expressions

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

### 3.3. Method Security Expressions