* We can also implement UserDetailsService in authentication-provider, to check the user from database like below
* For session timeout we need to configure

<session-config>

<session-timeout>1</session-timeout>

</session-config>

* To remember the user, instead of each time login, we need to use

<remember-me key=”myKey”/> inside http tag.

Also in the form we need to have check box with below data :

<input type=”checkbox” id=”remember” name=”\_spring\_security\_remember\_me” value=”true”/>

* We can also use UserDetailsService from spring for implemention of authentication provider using the tag

<security:authentication-manager>

<security:authentication-provider user-service-ref="myAppUserDtlsSrvc"/>

</security:authentication-manager>

The reference in the above tag is the implementation of UserDetailsService interface, which has a method to load the User and authorities of the user.

* When we implement the custom login page the following details must be followed :

j\_spring\_security\_check

j\_username

j\_password

and the form should be POST

To display the last exception we can use

SPRING\_SECURITY\_LAST\_EXCEPTION

* We can also configure the spring security using the jdbc-user-service and providing the datasource to it, here spring will fetch the data from the tables which are predefined by the spring.

<security:authentication-manager>

<security:authentication-provider>

<security:jdbc-user-service data-source-ref="dbDataSource" authorities-by-username-query="select USERNAME, ROLE from MYAPP\_USERS\_ROLES where USERNAME = ?"

users-by-username-query="select USERNAME, PASSWORD, CASE ENABLED WHEN 'Y' THEN '1' ELSE '0' END from MYAPP\_USERS where USERNAME = ?"/>

</security:authentication-provider>

</security:authentication-manager>

If users and authorities are not given default values are given and tables need to be created as required by spring framework.

* We can also use JDbcDAOImpl to set it to authentication provider user service ref by using the tag

<security:authentication-manager>

<security:authentication-provider user-service-ref="jdbcDaoImplObj"/>

</security:authentication-manager>

And

<bean id="jdbcDaoImplObj" class="org.springframework.security.core.userdetails.jdbc.JdbcDaoImpl">

<property name="dataSource" ref="dbDataSource" />

<property name="usersByUsernameQuery" value="select USERNAME, PASSWORD, CASE ENABLED WHEN 'Y' THEN '1' ELSE '0' END from MYAPP\_USERS where USERNAME = ?"></property>

<property name="authoritiesByUsernameQuery" value="select USERNAME, ROLE from MYAPP\_USERS\_ROLES where USERNAME = ?"/>

</bean>

Here class JdbcDaoImpl extends JdbcDaoSupport implements UserDetailsService.

* We should store password using the hash forms rather than normal text form, this can be done using the below tag in authentication provider

<security:authentication-manager>

<security:authentication-provider user-service-ref="jdbcDaoImplObj">

<security:password-encoder ref="bcryptEncoder" />

</security:authentication-provider>

</security:authentication-manager>

<bean name="bcryptEncoder" class="org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder"/>

* Requires channel in spring intercept url, when a request comes with http, if this property is set then it gets redirected to the channel automatically, valid values are http, https, any and the attribute is requires-channel
* Port mapping can also be used for the application for redirection purpose using the tags

<security:port-mappings>

<security:port-mapping http="8445" https="8094"/>

</security:port-mappings>

* Session Management
* Invalid sessions can be redirected to a particular url using below tag

<security:session-management invalid-session-url="/login/sessioninvalidated">

Inside http tag.

* When a user logs in to the browser which was not closed, which has the session invalid previously then new login might not be correct as the previous sessions JSESSIONID still may be present in the browser cookies, to delete we need to place below tag in logout.

<logout delete-cookies="JSESSIONID" />

* To disable concurrency login then use this in web.xml

<listener>

<listener-class>

org.springframework.security.web.session.HttpSessionEventPublisher

</listener-class>

</listener>

And

<session-management>

<concurrency-control max-sessions="1" />

</session-management>

Which will make first login invalid, and second login valid.

To make it viceversa

<session-management>

<concurrency-control max-sessions="1" error-if-maximum-exceeded="true" />

</session-management>

To redirect session expiry use

<security:concurrency-control max-sessions="1" error-if-maximum-exceeded="true" expired-url="/login/sessionexpired"/>

* Session fixation is done using session-fixation-protection attribute on <session-management>
* Method level security

To enable this we need to use the below tag

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

* The above tag will be used for access by using spring native @security annotation

Ex : @Secured("ROLE\_ADMIN")

Student getStudent(int studentId);

@Secured("IS\_AUTHENTICATED\_ANONYMOUSLY")

void saveStudent(Student student);

The above values will be sent to AccessDecisionManager for actual decision.

* To enable jsr250 annotations we better use this tag

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

* To enable pre-post annotations we use

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

Ex:

@PreAuthorize("hasAuthority('ROLE\_ADMIN')")

@PreAuthorize("isAnonymous()")

* We can enable more than one type of annotations at a time, but when we use on top of an method or interface only one type will be used.
* We can also use expression inside the annotations.
* We can use protect-pointcut to enable protection to certain methods at a time

<global-method-security>

<protect-pointcut expression="execution(\* com.mycompany.\*Service.\*(..))"

access="ROLE\_USER"/>

</global-method-security>

* Http element tag is used as below

<!-- Stateless RESTful service using Basic authentication -->

<http pattern="/restful/\*\*" create-session="stateless">

<intercept-url pattern='/\*\*' access='ROLE\_REMOTE' />

<http-basic />

</http>

<!-- Empty filter chain for the login page -->

<http pattern="/login.htm\*" security="none"/>

<!-- Additional filter chain for normal users, matching all other requests -->

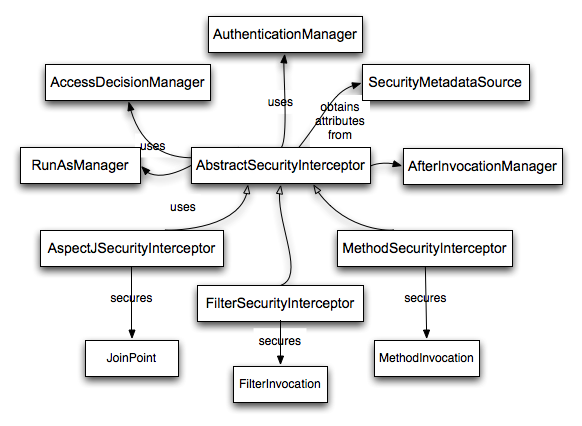
<http>

<intercept-url pattern='/\*\*' access='ROLE\_USER' />

<form-login login-page='/login.htm' default-target-url="/home.htm"/>

<logout />

</http>

* 
* We have BasicAuthenticationFilter and DigestAuthenticationFilter, in basic we have credentials are sent to server in plain text whereas in digest we have them in encrypted format.
* In the form we need to include below

<input type="hidden" name="${\_csrf.parameterName}" value="${\_csrf.token}"/>

And

<csrf /> inside the http tag.

* For json and ajax calls use the below requests use the crsf token in request headers

<head>

<meta name="\_csrf" content="${\_csrf.token}"/>

<!-- default header name is X-CSRF-TOKEN -->

<meta name="\_csrf\_header" content="${\_csrf.headerName}"/>

<!-- ... -->

</head>

For ajax calls use

$(function () {

var token = $("meta[name='\_csrf']").attr("content");

var header = $("meta[name='\_csrf\_header']").attr("content");

$(document).ajaxSend(function(e, xhr, options) {

xhr.setRequestHeader(header, token);

});

});

* Security headers
* Default security headers provided by spring security are

Cache Control

Content Type Options

HTTP Strict Transport Security

X-Frame-Options

X-XSS-Protection

* These headers are enabled using the tag <headers/> in http tag.
* Example to specify the headers in particular we can use

<http>

<!-- ... -->

<headers>

<cache-control />

<content-type-options />

<hsts />

<frame-options />

<xss-protection />

</headers>

</http>

* Cache Control

When this is enable spring security will add below headers

Cache-Control: no-cache, no-store, max-age=0, must-revalidate

Pragma: no-cache

Expires: 0

If we want cache specific response we can use HttpServletResponse.setHeader(String,String);

In spring mvc we can do it by

@EnableWebMvc

public class WebMvcConfiguration extends WebMvcConfigurerAdapter {

@Override

public void addResourceHandlers(ResourceHandlerRegistry registry) {

registry

.addResourceHandler("/resources/\*\*")

.addResourceLocations("/resources/")

.setCachePeriod(31556926);

}

// ...

}

* Content Type Options

When a resource is sent from server to client, its content-type is also should be sent, otherwise browser will sniff to check the content, this also lead to attacks, to disable the browser to sniff into the resource we use this property

<http>

<!-- ... -->

<headers>

<content-type-options />

</headers>

</http>

The header will look like

X-Content-Type-Options: nosniff

* HTTP Strict Transport Security

This makes sure the request which are coming to the server are https only so the man in middle attack can be avoided.

Strict-Transport-Security: max-age=31536000 ; includeSubDomains

Important Points

1. When Is The Session Created?

We can control exactly when our session gets created and how Spring Security will interact it:

always - a session will always be created if one doesn’t already exist

ifRequired - a session will be created only if required (default)

never - the framework will never create a session itself but it will use one if it already exists

stateless - no session will be created or used by Spring Security

<http create-session="ifRequired">...</http>

It’s very important to understand that this configuration only controls what Spring Security does – not the entire application. Spring Security may not create the session if we instruct it not to, but our app may!

By default, Spring Security will create a session when it needs one – this is “ifRequired“.

For a more stateless application, the “never” option will will ensure that Spring Security itself will not create any session; however, if the application creates one, then Spring Security will make use of it.

Finally, the strictest session creation option – “stateless” – is a guarantee that the application will not create any session at all.

This was introduced in Spring 3.1 and will effectively skip parts of the Spring Security filter chain – mainly the session related parts such as HttpSessionSecurityContextRepository, SessionManagementFilter, RequestCacheFilter.

These more strict control mechanism have the direct implication that cookies are not used and so each and every request needs to be re-authenticated. This stateless architecture plays well with REST APIs and their Statelessness constraint. They also work well with authentication mechanisms such as Basic and Digest Authentication.

2. Under The Hood

Before executing the Authentication process, Spring Security will run a filter responsible with storing the Security Context between requests – the SecurityContextPersistenceFilter. The context will be stored according to a strategy – HttpSessionSecurityContextRepository by default – which uses the HTTP Session as storage.

For the strict create-session=”stateless” attribute, this strategy will be replaced with another – NullSecurityContextRepository – and no session will be created or used to keep the context.

3. Concurrent Session Control

When a user that is already authenticated tries to authenticate again, the application can deal with that event in one of a few ways. It can either invalidate the active session of the user and authenticate the user again with a new session, or allow both sessions to exist concurrently.

First step in enabling the concurrent session-control support is to add the following listener in the web.xml:

<listener>

<listener-class>

org.springframework.security.web.session.HttpSessionEventPublisher

</listener-class>

</listener>

This is essential to make sure that the Spring Security session registry is notified when the session is destroyed.

To enable the scenario which allows multiple concurrent sessions for the same user the <session-management> element should be used in the XML configuration:

<http ...>

<session-management>

<concurrency-control max-sessions="2" />

</session-management>

</http>

4. Session Timeout

After the session has timed out, if the user sends a request with an expired session id, they will be redirected to an URL configurable via the namespace:

<session-management>

<concurrency-control expired-url="/sessionExpired.html" ... />

</session-management>

Similarly, if the user sends a request with a session id which is not expired, but entirely invalid, they will also be redirected to a configurable URL:

<session-management invalid-session-url="/invalidSession.html">

...

</session-management>

5. Prevent using URL Parameters for Session Tracking

Exposing session information in the URL is a growing security risk (from place 7 in 2007 to place 2 in 2013 on the OWASP Top 10 List).

Starting with Spring 3.0, the URL rewriting logic that would append the jsessionid to the URL can now be disabled by setting the disable-url-rewriting=”true” in the <http> namespace.

Alternatively, starting with Servlet 3.0, the session tracking mechanism can also be configured in the web.xml:

<session-config>

<tracking-mode>COOKIE</tracking-mode>

</session-config>

And programmatically:

servletContext.setSessionTrackingModes(EnumSet.of(SessionTrackingMode.COOKIE));

This chooses where to store the JSESSIONID – in the cookie or in a URL parameter.

6. Session Fixation Protection with Spring Security

The framework offers protection against typical Session Fixation attacks by configuring what happens to an existing session when the user tries to authenticate again:

<session-management session-fixation-protection="migrateSession"> ...

By default, Spring Security has this protection enabled (“migrateSession“) – on authentication a new HTTP Session is created, the old one is invalidated and the attributes from the old session are copied over.

If this is not the desired behaviour, two other options are available:

when “none” is set, the original session will not be invalidated

when “newSession” is set, a clean session will be created without any of the attributes from the old session being copied over

7. Working with the Session

7.1. Session Scoped Beans

A bean can be defined with session scope simply by using the @Scope annotation on beans declared in the web Context:

@Component

@Scope("session")

public class Foo { .. }

Or with XML:

<bean id="foo" scope="session"/>

Then, the bean can simply be injected into another bean:

@Autowired

private Foo theFoo;

And Spring will bind the new bean to the lifecycle of the HTTP Session.

7.2. Injecting the Raw Session into a Controller

The raw HTTP Session can also be injected directly into a Controller method:

@RequestMapping(..)

public void fooMethod(HttpSession session) {

session.addAttribute(Constants.FOO, new Foo();

...

Foo foo = (Foo) session.getAttribute(Constants.Foo);

}

7.3. Obtaining the Raw Session

The current HTTP Session can also be obtained programatically via the raw Servlet API:

ServletRequestAttributes attr = (ServletRequestAttributes)

RequestContextHolder.currentRequestAttributes();

HttpSession session= attr.getRequest().getSession(true); // true == allow create

Spring security 3.1

* Spring security is known as Acegi Security
* Does Authentication and Authorization
* Authentication

We have here Database, LDAP, OpenID, CAS-Central Authentication Service, Pre-Authentication (Siteminder), custom,

* Authorization

We have here URL based, Method Based (AOP) for service layer.

* Filters play a key role in spring security

<filter>

<filter-name>filter1</filter-name>

<filter-class>Filter1</filter-class>

</filter>

<filter-mapping>

<filter-name>filter1</filter-name>

<url-pattern>/\*</url-pattern>

</filter-mapping>

Here all the requests are intercepted by the filter Filter1, let us assume the Filter1 has a method doFilter().

* doFilter() method has parameters as ServletRequest, ServletResponse and FilterChain
* Filters will only intercept http request urls only not the forward and others to make the filter

To intercept use this

<filter>

<filter-name>filter1</filter-name>

<filter-class>Filter1</filter-class>

</filter>

<filter-mapping>

<filter-name>filter1</filter-name>

<url-pattern>/\*</url-pattern>

<dispatcher>REQUEST</dispatcher>

<dispatcher>FORWARD</dispatcher>

</filter-mapping>

* All the dispatcher types are

REQUEST

FORWARD

INCLUDE

ERROR

* Filter Chain is nothing but invoking the rest of the filters i.e

Public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) {

filterChain.doFilter(request, response);//if we have another filter in web.xml

servlet.service(request,response); //if we don’t have any filter left

}

* In a web application ContextLoadListener will create an applicationcontext using the configuration file mentioned in the web.xml, and then using the application context the bean of the Filter is created using the parameter in web.xml as mentioned below

// init ApplicationContext

XmlWebApplicationContext applicationContext = new XmlWebApplicationContext();

applicationContext.setConfigLocation("/WEB-INF/spring/\*.xml");

applicationContext.refresh();

// Use ApplicationContext

Filter filter = applicationContext.getBean("springSecurityFilterChain", Filter.class);

* Filter mapping order matters, so this should be present at the top in the web.xml.
* Here FilterChainProxy is created by DelegatingFilterProxy , filterchainproxy will call the required

Filter based upon the request, here is the pseudo code for the same:

public void doFilter(ServletRequest request, ServletResponse response, FilterChain filterChain) … {

Filter[] delegates = lookupDelegates(request);

for(Filter delegate : delegates) {

delegate.doFilter(request, response, chain);

if (delegate does not invoke filterChain.doFilter)

return;

}

filterChain.doFilter(request, response);

}

* When a request comes, it goes to DelegatingFilterProxy from where the work is delegated to FilterChainProxy, this will invoke the filter as required, as for the first time,

FilterChainProxy will invoke ExceptionTranslationFilter will see the request and try to invoke the application i.e rest of the filter as

try {

filterChain.doFilter(...);

}catch(AccessDeniedException e){

}

Which in turn ll call FilterSecurityInterceptor, this will check the requested url with the interceptor present in the security configuration file, if matched then ll check the access, if not present then throws an exception is ExceptionTranslationFilter, where it is catched as shown in above pseudo code.

catch(AccessDeniedException e){

// save HTTP request with RequestCache

// send to log in page w/

// AuthenticationEntryPoint

}

* See the above code the request is saved in RequestCache for redirecting after login, and after saving we will send the user to login page using Spring AuthenticationEntryPoint , which will send /spring\_security\_login to the user.
* Spring uses ant patterns

• Query parameters are not included in the match

• The context path is not included in the match

• ? matches one character

• \* matches zero or more characters (not a directory delimiter i.e. /)

• \*\* matches zero or more 'directories' in a path

* When a request comes with /spring\_security\_login then DelegationFilter🡪FilterChainProxy will call DefaultLoginPageGeneratingFilter which wll generate the login page and send the same to the user as response.
* When user fills the username and password and sends to j\_spring\_security\_check, then DelegatingFilterProxy 🡪FilterChainProxy 🡪 SecurityContextPersistentFilter, which see the session and check whether their is an authentication to it, if not it ignores it., then forwarded to UsernamePasswordAuthenticationFilter.
* The UsernamePasswordAuthenticationFilter, will call the AuthenticationManager which will validate the username and password sent by the filter and returns the User object for that user to the filter, this will again place the User object in SecurityContextHolder (This is where the spring checks whether the user is authenticated, if it is present in this then it means it got authenticated), then the filter will redirect the user to the saved request in RequestCache without the original parameters in the request.
* Then the request goes to SecurityContextPersistentFilter which saves the HttpSession of that user and clears the SecurityContextHolder.
* When the user sends the request after authentication, the SecurityContextHolder see the request and checks the session, if it authenticated then it saves the User object to SecurityContextHolder and redirects the request RequestCacheAwareFilter, here the previously saved request is sent to next filter instead of redirect request, so that the parameters will remain the request.
* Now FilterSecurityInterceptor sees the request and this will check the requested url with the interceptor present in the security configuration file, if matched then ll check the access. And returns to SecurityContextPersistent filter.
* Filters are like controllers, so we want to implement filter implement them in controllers.
* Filterchains will be different for different http elements in security configuration file.
* Generic pattern should be at the bottom.
* Spring security tag lib for jsp

<sec:authorize access=”authenticated”>

</sec:authorize>

The contents are only displayed if the user displaying is authenticated

<sec:authentication property=”name”/> -- displays the name

* Spring can’t differentiate between [www.xyz.com/1](http://www.xyz.com/1) and [www.xyz.com/1.css](http://www.xyz.com/1.css) so be careful while revoking the access to the css files by mentioning security=none.
* In spring security method level security can be enabled by

<security:global-method-security pre-post-annotations=”enabled”/>

* Whenever a method level security annotations are placed, spring will create a proxy class of it and the code will be like this

public class SecureMessageRepository implements MessageRepository {

public Message findOne(Long id) {

// PreAuthorize checks

Message result = delegate.findOne(id);

// PostAuthorize checks

return result;

}

…

// delegate = original MessageRepository

private MessageRepository delegate;

}

Ex: @PreAuthorize(“hasRole(‘ROLE\_USER’)”)

* We can also use spring expression language in method level security.

Spring security 3.2

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*