Very good set of articles about Spting & REST.

**Securing a RESTful Web Service with Spring Security 3.1**

## ****1. Overview****

This is the **third** of a series of articles about setting up a secure RESTful Web Service using Spring 3.1 and Spring Security 3.1 with Java based configuration. This article will focus on the security configuration using **Spring Security 3.1**, assuming some understanding of Spring Security basics and focusing on the specifics of **securing the RESTful web service**.

The REST with Spring series:

* **Part 1** – [Bootstrapping a web application with Spring 3.1 and Java based Configuration](http://www.baeldung.com/2011/10/20/bootstraping-a-web-application-with-spring-3-1-and-java-based-configuration-part-1/)
* **Part 2** – [Building a RESTful Web Service with Spring 3.1 and Java based Configuration](http://www.baeldung.com/2011/10/25/building-a-restful-web-service-with-spring-3-1-and-java-based-configuration-part-2/)
* **Part 4** – [RESTful Web Service Discoverability](http://www.baeldung.com/2011/11/06/restful-web-service-discoverability-part-4/)
* **Part 5** – [REST Service Discoverability with Spring](http://www.baeldung.com/2011/11/13/rest-service-discoverability-with-spring-part-5/)
* **Part 6 –** [Basic and Digest authentication for a RESTful Service with Spring Security 3.1](http://www.baeldung.com/2011/11/20/basic-and-digest-authentication-for-a-restful-service-with-spring-security-3-1/)
* **Part 7 –** [REST Pagination in Spring](http://www.baeldung.com/2012/01/18/rest-pagination-in-spring/)
* **Part 8** – [Authentication against a RESTful Service with Spring Security](http://www.baeldung.com/2012/12/20/authentication-against-a-restful-service/)
* **Part 9** – [ETags for REST with Spring](http://www.baeldung.com/2013/01/11/etags-for-rest-with-spring/)

## ****2. Introducing Spring Security in the**** web.xml

The architecture of Spring Security is based entirely on servlet filters and, as such, comes before Spring MVC in regards to the processing of HTTP requests. Keeping this in mind, to begin with, a **filter** needs to be declared in the web.xml of the application:

[?](http://www.baeldung.com/2011/10/31/securing-a-restful-web-service-with-spring-security-3-1-part-3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | <filter>     <filter-name>springSecurityFilterChain</filter-name>     <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>  </filter>  <filter-mapping>     <filter-name>springSecurityFilterChain</filter-name>     <url-pattern>/\*</url-pattern>  </filter-mapping> |

The filter must necessarily be named ‘springSecurityFilterChain’  to match the default bean created by Spring Security in the container.

Note that the defined filter is not the actual class implementing the security logic but a DelegatingFilterProxy with the purpose of delegating the Filter’s methods to an internal bean. This is done so that the target bean can still benefit from the Spring context lifecycle and flexibility.

The URL pattern used to configure the Filter is **/\*** even though the entire web service is mapped to **/api/\*** so that the security configuration has the option to secure other possible mappings as well, if required.

## ****3. The security configuration****

[?](http://www.baeldung.com/2011/10/31/securing-a-restful-web-service-with-spring-security-3-1-part-3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | <?xml version="1.0" encoding="UTF-8"?>  <beans:beans     xmlns="<http://www.springframework.org/schema/security>"     xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"     xmlns:beans="<http://www.springframework.org/schema/beans>"     xmlns:sec="<http://www.springframework.org/schema/security>"     xsi:schemaLocation="  <http://www.springframework.org/schema/security> <http://www.springframework.org/schema/security/spring-security-3.1.xsd>  <http://www.springframework.org/schema/beans> <http://www.springframework.org/schema/beans/spring-beans-3.1.xsd>">     <http entry-point-ref="restAuthenticationEntryPoint">        <intercept-url pattern="/api/admin/\*\*" access="ROLE\_ADMIN"/>        <custom-filter ref="myFilter" position="FORM\_LOGIN\_FILTER"/>        <logout />     </http>     <beans:bean id="myFilter" class=      "org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter">        <beans:property name="authenticationManager" ref="authenticationManager"/>        <beans:property name="authenticationSuccessHandler" ref="mySuccessHandler"/>     </beans:bean>     <beans:bean id="mySuccessHandler"      class="org.rest.security.MySavedRequestAwareAuthenticationSuccessHandler"/>     <authentication-manager alias="authenticationManager">        <authentication-provider>           <user-service>              <user name="temporary" password="temporary" authorities="ROLE\_ADMIN"/>              <user name="user" password="user" authorities="ROLE\_USER"/>           </user-service>        </authentication-provider>     </authentication-manager>  </beans:beans> |

Most of the configuration is done using the **security namespace** – for this to be enabled, the schema locations must be defined and pointed to the new 3.1 versions. The namespace is designed so that it expresses the common uses of Spring Security while still providing hooks to the underlying beans.

### ****3.1. The basics****

The <http> element is the main container element for HTTP security configuration. In the current implementation, it only secured a single mapping: /api/admin/\*\*. Note that the mapping is **relative to the root context** of the web application, not to the rest servlet; this is because the entire security configuration lives in the root Spring context and not in the child context of the servlet.

### ****3.2. The entry point****

In a standard web application, the authentication process may be automatically triggered when the client tries to access a secured resource without being authenticated – this is usually done by redirecting to a login page so that the user can enter credentials. However, for a **RESTful Web Service** this behavior doesn’t make much sense – authentication should only be done by a request to the correct URI and all other requests should simply fail with a **401 UNAUTHORIZED** status code if the user is not authenticated.

Spring Security handles this automatic triggering of the authentication process with the concept of an **entry point**; the entry point is a required part of the configuration, and can be injected via the entry-point-ref attribute of the <http> element. Keeping in mind that this functionality doesn’t make sense in the context of the RESTful web service, the new custom entry point is defined:

[?](http://www.baeldung.com/2011/10/31/securing-a-restful-web-service-with-spring-security-3-1-part-3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | @Component( "restAuthenticationEntryPoint" )  public final class RestAuthenticationEntryPoint implements AuthenticationEntryPoint{     @Override     public void commence( HttpServletRequest request, HttpServletResponse response,      AuthenticationException authException ) throws IOException{        response.sendError( HttpServletResponse.SC\_UNAUTHORIZED, "Unauthorized" );     }  } |

### ****3.3. The login****

There are multiple ways to do authentication for a RESTful Web Service – one of the default Spring Security provides is **form login** – which uses an authentication processing filter – org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter.

Since the <http> element doesn’t automatically create this particular filter by default, it needs to be explicitly specified in the configuration, using the <custom-filter> element at the position FORM\_LOGIN\_FILTER; the only required dependency for this bean is the authentication manager.

Note that for a standard web application, the **auto-config**attribute of the <http> element is shorthand syntax for some useful security configuration. While this may be appropriate for some very simple configurations, it doesn’t fit and should not be used for a REST API.

### ****3.4. Authentication should return 200 instead of 301****

By default, form login will answer a successful authentication request with a **301 MOVED PERMANENTLY** status code; this makes sense in the context of an actual login form which needs to redirect after login. For a RESTful web service however, the desired response for a successful authentication should be **200 OK**.

This is done by injecting a **custom authentication success handler** in the form login filter, to replace the default one. The new handler implements the exact same login as the default org.springframework.security.web.authentication.SavedRequestAwareAuthenticationSuccessHandler with one notable difference – the redirect logic is removed:

[?](http://www.baeldung.com/2011/10/31/securing-a-restful-web-service-with-spring-security-3-1-part-3/)

|  |  |
| --- | --- |
| 1  2  3  4 | // Use the DefaultSavedRequest URL  // final String targetUrl = savedRequest.getRedirectUrl();  // this.logger.debug( "Redirecting to DefaultSavedRequest Url: " + targetUrl );  // this.getRedirectStrategy().sendRedirect( request, response, targetUrl ); |

### ****3.5. The authentication manager and provider****

The authentication process uses an **in-memory provider** to perform authentication – this is meant to simplify the configuration as a production implementation of these artifacts is outside the scope of this post.

## ****4. Maven and other trouble****

In addition to the [pom.xml](https://gist.github.com/1299763) from the [first post](http://www.baeldung.com/2011/10/31/2011/10/20/bootstraping-a-web-application-with-spring-3-1-and-java-based-configuration-part-1/), as well as [the one](https://gist.github.com/1313645) from the [second post](http://www.baeldung.com/2011/10/25/building-a-restful-web-service-with-spring-3-1-and-java-based-configuration-part-2/), the Spring Security maven dependencies need to be added:

[?](http://www.baeldung.com/2011/10/31/securing-a-restful-web-service-with-spring-security-3-1-part-3/)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26 | <dependencies>     <dependency>        <groupId>org.springframework.security</groupId>        <artifactId>spring-security-web</artifactId>        <version>${spring-security.version}</version>     </dependency>     <dependency>        <groupId>org.springframework.security</groupId>        <artifactId>spring-security-config</artifactId>        <version>${spring-security.version}</version>     </dependency>     <dependency>        <groupId>org.springframework</groupId>        <artifactId>spring-tx</artifactId>        <version>${spring.version}</version>     </dependency>     <dependency>        <groupId>org.springframework</groupId>        <artifactId>spring-aop</artifactId>        <version>${spring.version}</version>     </dependency>  </dependencies>  <properties>     <spring-security.version>3.1.2.RELEASE</spring-security.version>     <spring.version>3.1.2.RELEASE</spring.version>  </properties> |

Notice that the security artifacts define **dependencies to the 3.0.x** versions of Spring, more specifically spring-security-web depends on spring-aop and on spring-tx version 3.0.x instead of the expected 3.1.x.

To understand why this is a problem, we need to understand how the **Maven conflict resolution algorithm** works – in case of conflict, Maven will chose which jar to include based on the distance between the particular dependency and the root of the tree. In our case, the conflicts are the spring-aop and spring-tx jars, appearing once with version 3.0.6 and once with 3.1.0. In the case of spring-aop, it appears once as a level 1 dependency of both spring-security-web and spring-security-config with version 3.0.6 , and once as a level 2 dependency of spring-webmvc with the version 3.1.0; since the 3.0.6 versioned jar is closer to the root, it will be the one chosen by the conflict resolution mechanism.

Now that we understand why it is that Maven will deploy the 3.0.6 version of the jars with the application and not the intended 3.1.0 version, we need to address the issue. The **solution** is to add the two dependencies, with the intended 3.1.0 versions, directly into the pom – this will shorten the distance between them and the root to 0 and will force Maven to use them first.

## ****5. Conclusion****

This post covered the basic security configuration and implementation for a RESTful service using **Spring Security 3.1**, discussing the web.xml, the security configuration, the HTTP status codes for the authentication process and the Maven resolution of the security artifacts. In the next articles I will focus on a Java based configuration for Spring Security, integration testing of the secure API using the [rest-assured](http://code.google.com/p/rest-assured/) library and HTTP basic authentication. In the meantime, check out the [github project](https://github.com/eugenp/REST#readme).