Table of Contents

Bootstrap a Web Application with Spring 4

Return to Content

Contents

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

Table of Contents

1. Overview

The tutorial illustrates how to **Bootstrap a Web Application with Spring** and also discusses how to make the jump **from XML to Java** without having to completely migrate the entire XML configuration.

2. The Maven pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="
 http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>org</groupId>
 <artifactId>rest</artifactId>
 <version>0.0.1-SNAPSHOT</version>
 <packaging>war</packaging>

 <dependencies>

 <dependency>
 <groupId>org.springframework</groupId>
 <artifactId>spring-webmvc</artifactId>
 <version>${spring.version}</version>
 <exclusions>
 <exclusion>
 <artifactId>commons-logging</artifactId>
 <groupId>commons-logging</groupId>
 </exclusion>
 </exclusions>
 </dependency>

 </dependencies>

 <build>
 <finalName>rest</finalName>

 <plugins>
 <plugin>
 <groupId>org.apache.maven.plugins</groupId>
 <artifactId>maven-compiler-plugin</artifactId>
 <version>3.1</version>
 <configuration>
 <source>1.6</source>
 <target>1.6</target>
 <encoding>UTF-8</encoding>
 </configuration>
 </plugin>
 </plugins>
 </build>

 <properties>
 <spring.version>4.0.5.RELEASE</spring.version>
 </properties>

</project>
```

2.1. The cglib dependency before Spring 3.2

You may wonder why *cglib* is a dependency – it turns out there is a valid reason to include it – the entire configuration cannot function without it. If removed, Spring will throw:

Caused by: java.lang.IllegalStateException: CGLIB is required to process @Configuration classes. Either add CGLIB to the classpath or remove the following @Configuration bean definitions

The reason this happens is explained by the way Spring deals with *@Configuration* classes. These classes are effectively beans, and because of this they need to be aware of the Context, and respect scope and other bean semantics. This is achieved by dynamically creating a cglib proxy with this awareness for each *@Configuration* class, hence the cglib dependency.

Also, because of this, there are a few restrictions for *Configuration* annotated classes:

- Configuration classes **should not be final**
- They should have a constructor with no arguments

2.2. The cglib dependency in Spring 3.2 and beyond

Starting with Spring 3.2, it is **no longer necessary to add cglib as an explicit dependency**. This is because Spring is in now inlining *cglib* – which will ensure that all class based proxying functionality will work out of the box with Spring 3.2.

The new cglib code is placed under the Spring package: *org.springframework.cglib* (replacing the original *net.sf.cglib*). The reason for the package change is to avoid conflicts with any *cglib* versions already existing on the classpath.

Also, the new cglib 3.0 is now used, upgraded from the older 2.2 dependency (see this JIRA issue [https://jira.springsource.org/browse/SPR-9669] for more details).

Finally, now that Spring 4.0 is out in the wild, changes like this one (removing the cglib dependency) are to be expected with Java 8 just around the corner – you can watch this Spring Jira [https://jira.springsource.org/browse/SPR-9639] to keep track of the Spring support, and the Java 8 Resources page [http://www.baeldung.com/java8] to keep tabs on the that.

3. The Java based Web Configuration

```
@Configuration
@ImportResource( { "classpath*:/rest_config.xml" } )
@ComponentScan( basePackages = "org.rest" )
@PropertySource({ "classpath:rest.properties", "classpath:web.properties" })
public class AppConfig{

 @Bean
Â Â  public static PropertySourcesPlaceholderConfigurer properties() {
Â Â  return new PropertySourcesPlaceholderConfigurer();
Â Â  }
}
```

First, the **@Configuration** annotation – this is the main artifact used by the Java based Spring configuration; it is itself meta-annotated with *@Component*, which makes the annotated classes **standard beans** and as such, also candidates for component scanning. The main purpose of *@Configuration* classes is to be sources of bean definitions for the Spring IoC Container. For a more detailed description, see the official docs [http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/beans.html#beans-java].

Then, **@ImportResource** is used to import the existing XML based Spring configuration. This may be configuration which is still being migrated from XML to Java, or simply legacy configuration that you wish to keep. Either way, importing it into the Container is essential for a successful migration, allowing small steps without to much risk. The equivalent XML annotation that is replaced is:

<import resource="classpath:/rest_config.xml" />*

Moving on to **@ComponentScan** – this configures the component scanning directive, effectively replacing the XML:

```
<context:component-scan base-package="org.rest" />
```

As of Spring 3.1, the *@Configuration* are excluded from classpath scanning by default – see this JIRA issue [https://jira.springsource.org/browse/SPR-8808]. Before Spring 3.1 though, these classes should have been excluded explicitly:

```
excludeFilters = { @ComponentScan.Filter( Configuration.class ) }
```

The *@Configuration* classes should not be autodiscovered because they are already specified and used by the Container – allowing them to be rediscovered and introduced into the Spring context will result in the following error:

Caused by: org.springframework.context.annotation.ConflictingBeanDefinitionException: Annotation-specified bean name 'webConfig' for bean class [org.rest.spring.AppConfig] conflicts with existing, non-compatible bean definition of same name and class [org.rest.spring.AppConfig]

And finally, using the **@Bean** annotation to configure the **properties support** – Â *PropertySourcesPlaceholderConfigurer* is initialized in a *@Bean* annotated method, indicating it will produce a Spring bean managed by the Container. This new configuration has replaced the following XML:

```
<context:property-placeholder
location="classpath:persistence.properties, classpath:web.properties"
ignore-unresolvable="true"/>
```

For a more in depth discussion on why it was necessary to manually register the *PropertySourcesPlaceholderConfigurer* bean, see the Properties with Spring Tutorial [http://www.baeldung.com/2012/02/06/properties-with-spring/].

3.1. The web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="
 http://java.sun.com/xml/ns/javaee"
Â Â  Â  xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
Â Â  Â  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
Â Â  Â xsi:schemaLocation="
 http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
Â Â  Â id="rest" version="3.0">

 <context-param>
 <param-name>contextClass</param-name>
 <param-value>
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext
 </param-value>
 </context-param>
 <context-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>org.rest.spring.root</param-value>
 </context-param>
 <listener>
       <listener-class>org.springframework.web.context.ContextLoaderListener</
listener-class>
 </listener>

 <servlet>
 <servlet-name>rest</servlet-name>
 <servlet-class>
 org.springframework.web.servlet.DispatcherServlet
 </servlet-class>
 <init-param>
 <param-name>contextClass</param-name>
 <param-value>
 org.springframework.web.context.support.AnnotationConfigWebApplicationContext
 </param-value>
 </init-param>
 <init-param>
 <param-name>contextConfigLocation</param-name>
 <param-value>org.rest.spring.rest</param-value>
 </init-param>
 <load-on-startup>1</load-on-startup>
 </servlet>
 <servlet-mapping>
 <servlet-name>rest</servlet-name>
 <url-pattern>/api/*</url-pattern>
 </servlet-mapping>
 <welcome-file-list>
 <welcome-file />
 </welcome-file-list>

</web-app>
```

First, the root context is defined and configured to use *AnnotationConfigWebApplicationContext* instead of the default *XmlWebApplicationContext*. The newer *AnnotationConfigWebApplicationContext* accepts *@Configuration* annotated classes as input for the Container configuration and is needed in order to set up the Java based context. Unlike *XmlWebApplicationContext*, it assumes no default configuration class locations, so the *"contextConfigLocation"init-param* for the Servlet must be set. This will point to the java package where the *@Configuration* classes are located; the fully qualified name(s) of the classes are also supported.

Next, the *DispatcherServlet* is configured to use the same kind of context, with the only difference that it's loading configuration classes out of a different package.

Other than this, the *web.xml* doesn't really change from a XML to a Java based configuration.

4. Conclusion

The presented approach allows for a smooth **migration of the Spring configuration** from XML to Java, mixing the old and the new. This is important for older projects, which may have a lot of XML based configuration that cannot be migrated all at once.

This way, in a migration, the XML beans can be ported in small increments.

In the next article on REST with Spring [http://www.baeldung.com/2011/10/25/building-a-restful-web-service-with-spring-3-1-and-java-based-configuration-part-2/], I cover setting up MVC in the project, configuration of the HTTP status codes, payload marshalling and content negotiation.

The implementation of this *Bootstrap a Spring Web App Tutorial* can be downloaded as a working sample project. [https://my.leadpages.net/leadbox/147e9e473f72a2%3A13a71ac76b46dc/5745710343389184/]

This is an Eclipse based project, so it should be easy to import and run as it is.

Â

http://twitter.com/share

java [http://www.baeldung.com/tag/java-2/], Spring [http://www.baeldung.com/tag/spring/]

Build a REST API with Spring 4 and Java Config

Return to Content

Contents

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

Â

Table of Contents

1. Overview

This article shows how to **set up REST in Spring** – the Controller and HTTP response codes, configuration of payload marshalling and content negotiation.

2. Understanding REST in Spring

The Spring framework supports 2 ways of creating RESTful services:

- using MVC with *ModelAndView*
- using HTTP message converters

The **ModelAndView** approach is older and much better documented, but also more verbose and configuration heavy. It tries to shoehorn the REST paradigm into the old model, which is not without problems. The Spring team understood this and provided first-class REST support starting with **Spring 3.0**.

The new approach, based on **HttpMessageConverter**and **annotations**, is much more lightweight and easy to implement. Configuration is minimal and it provides sensible defaults for what you would expect from a RESTful service. It is however newer and a a bit on the light side concerning documentation; what's , the reference doesn't go out of it's way to make the distinction and the tradeoffs between the two approaches as clear as they should be. Nevertheless, this is the way RESTful services should be build after Spring 3.0.

3. The Java configuration

http://www.baeldung.com/wp-content/uploads/2013/11/SPRING-JAVA-CONFIGURATION.jpg

```
@Configuration
@EnableWebMvc
public class WebConfig{
 //
}
```

The new **@EnableWebMvc** annotation does a number of useful things – specifically, in the case of REST, it detect the existence of Jackson and JAXB 2 on the classpath and automatically creates and registers default **JSON and XML converters**. The functionality of the annotation is equivalent to the XML version:

<mvc:annotation-driven />

This is a shortcut, and though it may be useful in many situations, it's not perfect. When more complex configuration is needed, remove the annotation and extend *WebMvcConfigurationSupport* directly.

4. Testing the Spring context

Starting with **Spring 3.1**, we get [http://spring.io/blog/2011/06/21/spring-3-1-m2-testing-with-configuration-classes-and-profiles/] first-class testing support for *@Configuration* classes:

```
@RunWith( SpringJUnit4ClassRunner.class )
@ContextConfiguration( classes = { ApplicationConfig.class,
 PersistenceConfig.class },
 loader = AnnotationConfigContextLoader.class )
public class SpringTest{

 @Test
 public void whenSpringContextIsInstantiated_thenNoExceptions(){
 // When
 }
}
```

The Java configuration classes are simply specified with the **@ContextConfiguration** annotation and the new *AnnotationConfigContextLoader* loads the bean definitions from the *@Configuration* classes.

Notice that the *WebConfig* configuration class was not included in the test because it needs to run in a Servlet context, which is not provided.

5. The Controller

The **@Controller** is the central artifact in the entire Web Tier of the RESTful API. For the purpose of this post, the controller is modeling a simple REST resource – *Foo*:

```
@Controller
@RequestMapping( value = "/foos" )
```

```
class FooController{

 @Autowired
 IFooService service;

 @RequestMapping( method = RequestMethod.GET )
 @ResponseBody
 public List< Foo > findAll(){
 return service.findAll();
 }

 @RequestMapping( value = "/{id}", method = RequestMethod.GET )
 @ResponseBody
 public Foo findOne( @PathVariable( "id" ) Long id ){
 return RestPreconditions.checkFound( service.findOne( id ) );
 }

 @RequestMapping( method = RequestMethod.POST )
 @ResponseStatus( HttpStatus.CREATED )
 @ResponseBody
 public Long create( @RequestBody Foo resource ){
 Preconditions.checkNotNull( resource );
 return service.create( resource );
 }

 @RequestMapping( value = "/{id}", method = RequestMethod.PUT )
 @ResponseStatus( HttpStatus.OK )
 public void update( @PathVariable( "id" ) Long id, @RequestBody Foo resource ){
 Preconditions.checkNotNull( resource );
 RestPreconditions.checkNotNull( service.getById( resource.getId() ) );
 service.update( resource );
 }

 @RequestMapping( value = "/{id}", method = RequestMethod.DELETE )
 @ResponseStatus( HttpStatus.OK )
 public void delete( @PathVariable( "id" ) Long id ){
 service.deleteById( id );
 }

}
```

You may have noticed I'm using a very simple, guava style *RestPreconditions* utility:

```
public class RestPreconditions {
 public static <T> T checkFound(final T resource) {
 if (resource == null) {
 throw new MyResourceNotFoundException();
 }
 return resource;
 }
}
```

The Controller implementation is **non-public** – this is because it doesn't need to be. Usually the controller is the last in the chain of dependencies – it receives HTTP requests from the Spring front controller (the *DispathcerServlet*) and simply delegate them forward to a service layer. If there is no use case where the controller has to be injected or manipulated through a direct reference, then I prefer not to declare it as public.

The **request mappings** are straightforward – as with any controller, the actual *value* of the mapping as well as the HTTP *method* are used to determine the target method for the request. ***@RequestBody*** will bind the parameters of the method to the body of the HTTP request, whereas **@ResponseBody** does the same for the response and return type. They also ensure that the resource will be marshalled and unmarshalled using the correct HTTP converter. **Content negotiation** will take place to choose which one of the active converters will be used, based mostly on the *Accept* header, although other HTTP headers may be used to determine the representation as well.

6. Mapping the HTTP response codes

The status codes of the HTTP response are one of the most important parts of the REST service, and the subject can quickly become very complex. Getting these right can be what makes or breaks the service.

6.1. Unmapped requests

If Spring MVC receives a request which doesn't have a mapping, it considers the request not to be allowed and returns a **405 METHOD NOT ALLOWED** back to the client. It is also good practice to include the **Allow HTTP header** when returning a *405* to the client, in order to specify which operations **are** allowed. This is the standard behavior of Spring MVC and does not require any additional configuration.

6.2. Valid, mapped requests

For any request that does have a mapping, Spring MVC considers the request valid and responds with **200 OK** if no other status code is specified otherwise. It is because of this that controller declares different ***@ResponseStatus*** for the *create*, *update* and *delete* actions but not for *get*, which should indeed return the default 200 OK.

6.3. Client error

In case of a **client error**, custom exceptions are defined and mapped to the appropriate error codes. Simply throwing these exceptions from any of the layers of the web tier will ensure Spring maps the corresponding status code on the HTTP response.

```
@ResponseStatus( value = HttpStatus.BAD_REQUEST )
public class BadRequestException extends RuntimeException{
 //
}
@ResponseStatus( value = HttpStatus.NOT_FOUND )
public class ResourceNotFoundException extends RuntimeException{
 //
}
```

These exceptions are part of the REST API and, as such, should only be used in the appropriate layers corresponding to REST; if for instance a DAO/DAL layer exist, it should not use the exceptions directly. Note also that these are not **checked exceptions** but **runtime exceptions** – in line with Spring practices and idioms.

6.4. Using @ExceptionHandler

Another option to map custom exceptions on specific status codes is to use the ***@ExceptionHandler*** annotation in the controller. The problem with that approach is that the annotation only applies to the controller in which it is defined, not to the entire Spring Container, which means that it needs to be declared in each controller individually. This quickly becomes cumbersome, especially in more complex applications which many controllers. There are a few **JIRA issues** opened with Spring at this time to handle this and other related limitations: SPR-8124 [https://jira.springsource.org/browse/SPR-8124], SPR-7278 [https://jira.springsource.org/browse/SPR-7278], SPR-8406 [https://jira.springsource.org/browse/SPR-8406].

7. Additional Maven dependencies

In addition to the *spring-webmvc* dependency required for the standard web application [http://www.baeldung.com/spring-with-maven#mvc], we'll need to set up content marshalling and unmarshalling for the REST API:

```
<dependencies>
 <dependency>
Â Â  <groupId>com.fasterxml.jackson.core</groupId>
 Â Â  <artifactId>jackson-databind</artifactId>
Â Â  <version>${jackson.version}</version>
 </dependency>
 <dependency>
 <groupId>javax.xml.bind</groupId>
 <artifactId>jaxb-api</artifactId>
 <version>${jaxb-api.version}</version>
 <scope>runtime</scope>
 </dependency>
</dependencies>

<properties>
 <jackson.version>2.4.0</jackson.version>
 <jaxb-api.version>2.2.11</jaxb-api.version>
</properties>
```

These are the libraries used to convert the representation of the REST resource to either **JSON** or **XML**.

8. Conclusion

This tutorial illustrated how to implement and configure a REST Service using Spring 4 and Java based configuration, discussing HTTP response codes, basic Content Negotiation and marshaling.

In the next articles of the series I will focus on Discoverability of the API [http://www.baeldung.com/2011/11/06/restful-web-service-discoverability-part-4/], advanced **content negotiation** and working with **additional representations** of a Resource.

The implementation of this *Spring REST API Tutorial* can be downloaded as a working sample project. [https://my.leadpages.net/leadbox/143ecbd73f72a2%3A13a71ac76b46dc/5762893836451840/]

This is an Eclipse based project, so it should be easy to import and run as it is.

http://twitter.com/share

java [http://www.baeldung.com/tag/java-2/], REST [http://www.baeldung.com/tag/rest/], Spring [http://www.baeldung.com/tag/spring/], testing [http://www.baeldung.com/tag/testing/]

Spring Security for a REST API

Return to Content

Contents

Table of Contents

1. Overview

This tutorial shows how to **Secure a REST Service using Spring and Spring Security 3.1** with Java based configuration. The article will focus on how to set up the Security Configuration specifically for the REST API using a Login and Cookie approach.

2. 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:

```
<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’*Â 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

```
<?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-3.2.xsd
 http://www.springframework.org/schema/beans
 http://www.springframework.org/schema/beans/spring-beans-4.0.xsd">

 <http entry-point-ref="restAuthenticationEntryPoint">
 <intercept-url pattern="/api/admin/**" access="ROLE_ADMIN"/>

 <form-login
 authentication-success-handler-ref="mySuccessHandler"
 authentication-failure-handler-ref="myFailureHandler"
 />

 <logout />
 </http>

 <beans:bean id="mySuccessHandler"
 class="org.rest.security.MySavedRequestAwareAuthenticationSuccessHandler"/>
 <beans:bean id="myFailureHandler"

>

 <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 correct 3.1 or 3.2 XSD versions. The namespace is designed so that it expresses the common uses of Spring Security while still providing hooks raw beans to accommodate more advanced scenarios. >> **Signup for my upcoming Video Course on Building a REST API with Spring 4** [http://products.baeldung.com/rest-api-with-spring]

3.1. The <http> element

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 **REST 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** – this 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 REST Service, the new custom entry point is defined to simply return 401 whenever it is triggered:

```
@Component( "restAuthenticationEntryPoint" )
public class RestAuthenticationEntryPoint implements AuthenticationEntryPoint{

 @Override
  public void commence( HttpServletRequest request, HttpServletResponse
 response,
 AuthenticationException authException ) throws IOException{
 response.sendError( HttpServletResponse.SC_UNAUTHORIZED, "Unauthorized" );
 }
}
```

A quick sidenote here is that the 401 is sent without the *WWW-Authenticate* header, as required by the HTTP Spec – we can of course set the value manually if we need to.

3.3. The Login Form for REST

There are multiple ways to do Authentication for a REST API – one of the default Spring Security provides is **Form Login** – which uses an authentication processing filter – *org.springframework.security.web.authentication.UsernamePasswordAuthenticationFilter*.

The *<form-login>* element will create this filter and will also allow us to set our custom authentication success handler on it. This can also be done manually by using the *<custom-filter>* element to register a filter at the position *FORM_LOGIN_FILTER* – but the namespace support is flexible enough.

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:

```
public class MySavedRequestAwareAuthenticationSuccessHandler
 extends SimpleUrlAuthenticationSuccessHandler {

 private RequestCache requestCache = new HttpSessionRequestCache();

 @Override
    public    void    onAuthenticationSuccess(HttpServletRequest    request,
 HttpServletResponse response,
 Authentication authentication) throws ServletException, IOException {
 SavedRequest savedRequest = requestCache.getRequest(request, response);

 if (savedRequest == null) {
 clearAuthenticationAttributes(request);
 return;
 }
 String targetUrlParam = getTargetUrlParameter();
 if (isAlwaysUseDefaultTargetUrl() ||
 (targetUrlParam != null &&
 StringUtils.hasText(request.getParameter(targetUrlParam)))) {
 requestCache.removeRequest(request, response);
 clearAuthenticationAttributes(request);
 return;
 }

 clearAuthenticationAttributes(request);
 }

 public void setRequestCache(RequestCache requestCache) {
 this.requestCache = requestCache;
 }
}
```

3.5. Failed Authentication should return 401 instead of 302

Similarly – we configured the authentication failure handler – same way we did with the success handler.

Luckily – in this case, we don't need to actually define a new class for this handler – the standard implementation – *SimpleUrlAuthenticationFailureHandler* – does just fine.

The only difference is that – now that we're defining this explicitly in our XML config – it's **not going to get a default defaultFailureUrl from Spring** – and so it won't redirect.

3.6. 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.

3.7 Finally – Authentication against the running REST Service

Now let's see how we can authenticate against the REST API – the URL for login isÂ `/` *j_spring_security_check* – and a simple *curl* command performing login would be:

```
curl -i -X POST -d j_username=user -d j_password=userPass
http://localhost:8080/spring-security-rest/j_spring_security_check
```

This request will return the Cookie which will then be used by any subsequent request against the REST Service.

We can use *curl* to authentication and **store the cookie it receives in a file**:

```
curl -i -X POST -d j_username=user -d j_password=userPass -c /opt/cookies.txt
http://localhost:8080/spring-security-rest/j_spring_security_check
```

Then **we can use the cookie from the file** to do further authenticated requests:

```
curl -i --header "Accept:application/json" -X GET -b /opt/cookies.txt
http://localhost:8080/spring-security-rest/api/foos
```

This authenticated request will correctly **result in a 200 OK**:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json;charset=UTF-8
Transfer-Encoding: chunked
Date: Wed, 24 Jul 2013 20:31:13 GMT

[{"id":0,"name":"JbidXc"}]
```

4. Maven and other trouble

The Spring core dependencies [http://www.baeldung.com/spring-with-maven#mvc] necessary for a web application and for the REST Service have been discussed in detail. For security, we'll need to add: *spring-security-web* and *spring-security-config* – all of these have also been covered in the Maven for Spring Security [http://www.baeldung.com/spring-security-with-maven] tutorial.

It's worth paying close attention to the way Maven will resolve the older Spring dependencies – the resolution strategy will start causing problems [http://www.baeldung.com/spring-security-with-maven#maven_problem] once the security artifacts are added to the pom. To address this problem, some of the core dependencies will need to be overridden in order to keep them at the right version.

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.

The implementation of this Spring Security REST Tutorial can be downloaded as a working sample project. [https://my.leadpages.net/leadbox/14476c373f72a2%3A13a71ac76b46dc/5664530495438848/]This is an Eclipse based project, so it should be easy to import and run as it is.

http://twitter.com/share

REST [http://www.baeldung.com/tag/rest/], security [http://www.baeldung.com/tag/security/], Spring [http://www.baeldung.com/tag/spring/]

Spring Security Basic Authentication

Return to Content

Contents

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

1. Overview

This tutorial shows how to set up, configure and customize **Basic Authentication with Spring**. We’re going to built on top of the simple Spring MVC example [http://www.baeldung.com/spring-mvc-tutorial], and secure the UI of the MVC application with the Basic Auth mechanism provided by Spring Security.

2. The Spring Security Configuration

The Configuration for Spring Security is still XML:

```
<?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"
 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-3.2.xsd">

 <http use-expressions="true">
 <intercept-url pattern="/**" access="isAuthenticated()" />

 <http-basic />
 </http>

 <authentication-manager>
 <authentication-provider>
 <user-service>
 <user name="user1" password="user1Pass" authorities="ROLE_USER" />
 </user-service>
 </authentication-provider>
 </authentication-manager>

</beans:beans>
```

This is one of the last pieces of configuration in Spring that still need XML – Java Configuration for Spring Security [https://github.com/SpringSource/spring-security-javaconfig] is still a work in progress.

What is relevant here is the *<http-basic>* element inside the main *<http>* element of the configuration – this is enough to enable Basic Authentication for the entire application. The Authentication Manager is not the focus of this tutorial, so we are using an in memory manager with the user and password defined in plaintext.

The *web.xml* of the web application enabling Spring Security has already been discussed in the Spring Logout tutorial [http://www.baeldung.com/spring-security-login#web_xml].

3. Consuming The Secured Application

The *curl* command is our go to tool for consuming the secured application.

First, let's try to request the */homepage.html* without providing any security credentials:

```
curl -i http://localhost:8080/spring-security-mvc-basic-auth/homepage.html
```

We get back the expected *401 Unauthorized* and the Authentication Challenge [http://tools.ietf.org/html/rfc1945#section-10.16]:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
Set-Cookie:     JSESSIONID=E5A8D3C16B65A0A007CFAACAEEE6916B;     Path=/spring-
security-mvc-basic-auth/; HttpOnly
WWW-Authenticate: Basic realm="Spring Security Application"
Content-Type: text/html;charset=utf-8
Content-Length: 1061
Date: Wed, 29 May 2013 15:14:08 GMT
```

The browser would interpret this challenge and prompt us for credentials with a simple dialog, but since we're using *curl*, this isn't the case.

Now, let's request the same resource – the homepage – but **provide the credentials** to access it as well:

```
curl -i --user user1:user1Pass http://localhost:8080/spring-security-mvc-basic-
auth/homepage.html
```

Now, the response from the server is *200 OK* along with a *Cookie*:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie:     JSESSIONID=301225C7AE7C74B0892887389996785D;     Path=/spring-
security-mvc-basic-auth/; HttpOnly
Content-Type: text/html;charset=ISO-8859-1
Content-Language: en-US
Content-Length: 90
Date: Wed, 29 May 2013 15:19:38 GMT
```

From the browser, the application can be consumed normally – the only difference is that a login page is no longer a hard requirement since all browsers support Basic Authentication and use a dialog to prompt the user for credentials.

4. Further Configuration – The Entry Point

By default, the *BasicAuthenticationEntryPoint* provisioned by Spring Security returns a full html page for a *401 Unauthorized* response back to the client. This html representation of the error renders well in a browser, but it not well suited for other scenarios, such as a REST API where a json representation may be preferred.

The namespace is flexible enough for this new requirement as well – to address this – the entry point can be overridden:

```
<http-basic entry-point-ref="myBasicAuthenticationEntryPoint" />
```

The new entry point is defined as a standard bean:

```
@Component
public class MyBasicAuthenticationEntryPoint extends
 BasicAuthenticationEntryPoint {

 @Override
 public void commence
     (HttpServletRequest request, HttpServletResponse response,
 AuthenticationException authEx)
 throws IOException, ServletException {
  response.addHeader("WWW-Authenticate", "Basic realm=\"" + getRealmName() +
 "\"");
 response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);
 PrintWriter writer = response.getWriter();
 writer.println("HTTP Status 401 - " + authEx.getMessage());
 }

 @Override
 public void afterPropertiesSet() throws Exception {
 setRealmName("Baeldung");
 super.afterPropertiesSet();
 }
}
```

By writing directly to the HTTP Response we now have full control over the format of the response body.

5. The Maven Dependencies

The Maven dependencies for Spring Security have been discussed before in the Spring Security with Maven article [http://www.baeldung.com/spring-security-with-maven] – we will need both *spring-security-web* and *spring-security-config* available at runtime.

6. Conclusion

In this example we secured an MVC application with Spring Security and Basic Authentication. We discussed the XML configuration and we consumed the application with simple curl commands. Finally took control of the exact error message format – moving from the standard HTML error page to a custom text or json format.

The implementation of this SpringÂ tutorial can be found in the github project [https://github.com/eugenp/tutorials/tree/master/spring-security-basic-auth#readme] â€" this is an Eclipse based project, so it should be easy to import and run as it is. When the project runs locally, the sample html can be accessed at:

http://localhost:8080/spring-security-mvc-basic-auth/homepage.html

http://twitter.com/share

security [http://www.baeldung.com/tag/security/], Spring [http://www.baeldung.com/tag/spring/]

Spring Security Digest Authentication

Return to Content

Contents

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

1. Overview

This tutorial shows how to set up, configure and customize Digest Authentication with Spring. Similar to the previous article covering Basic Authentication [http://www.baeldung.com/spring-security-basic-authentication], weâ€™re going to built on top of the Spring MVC tutorial, and secure the application with the Digest Auth mechanism provided by Spring Security.

2. The Security XML Configuration

First thing to understand about the configuration is that, while Spring Security does have full out of the box support for the Digest authentication mechanism, this support is **not as well integrated into the namespace** as Basic Authentication was.

In this case, we need to manually **define the raw beans** that are going to make up the security configuration – the *DigestAuthenticationFilter* and the *DigestAuthenticationEntryPoint*:

```
<?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"
 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-3.2.xsd">

Â Â Â  <beans:bean id="digestFilter"
 class="org.springframework.security.web.authentication.www.DigestAuthenticationFilter">
Â Â Â Â Â Â Â  <beans:property name="userDetailsService" ref="userService" />
Â   Â   Â   Â   Â   Â   Â      <beans:property   name="authenticationEntryPoint"
 ref="digestEntryPoint" />
Â Â Â  </beans:bean>
Â Â Â  <beans:bean id="digestEntryPoint"Â
 class="org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint"
Â Â Â Â Â Â Â  <beans:property name="realmName" value="Contacts Realm via Digest
 Authentication" />
Â Â Â Â Â Â Â  <beans:property name="key" value="acegi" />
Â Â Â  </beans:bean>

 <!-- the security namespace configuration -->
 <http use-expressions="true" entry-point-ref="digestEntryPoint">
 <intercept-url pattern="/**" access="isAuthenticated()" />
 <custom-filter ref="digestFilter" after="BASIC_AUTH_FILTER" />
 </http>

 <authentication-manager>
 <authentication-provider>
 <user-service id="userService">
 <user name="user1" password="user1Pass" authorities="ROLE_USER" />
 </user-service>
 </authentication-provider>
 </authentication-manager>

</beans:beans>
```

Next, we need to integrate these beans into the overall security configuration – and in this case, the namespace is still flexible enough to allow us to do that.

The first part of this is pointing to the custom entry point bean, via the *entry-point-ref* attribute of the main *<http>* element.

The second part is **adding the newly defined digest filter into the security filter chain**. Since this filter is functionally equivalent to the *BasicAuthenticationFilter*, we are using the same relative position in the chain – this is specified by the *BASIC_AUTH_FILTER* alias in the overall Spring Security Standard Filters [http://static.springsource.org/spring-security/site/docs/3.1.x/reference/ns-config.html#ns-custom-filters].

Finally, notice that the Digest Filter is configured to **point to the user service bean** – and here, the namespace is again very useful as it allows us to specify a bean name for the default user service created by the *<user-service>* element:

```
<user-service id="userService">
```

3. Consuming the Secured Application

We're going to be using **the curl command** to consume the secured application and understand how a client can interact with it.

Let's start by requesting the homepage – **without providing security credentials** in the request:

```
curl -i http://localhost/spring-security-mvc-digest-auth/homepage.html
```

As expected, we get back a response with a *401 Unauthorized* status code:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
Set-Cookie:   JSESSIONID=CF0233C...;   Path=/spring-security-mvc-digest-auth/;
 HttpOnly
WWW-Authenticate: Digest  realm="Contacts  Realm  via  Digest  Authentication",
 qop="auth",
 nonce="MTM3MzYzODE2NTg3OTo3MmYxN2JkOWYxZTc4MzdmMzBiN2Q0YmY0ZTU0N2RkZg=="
Content-Type: text/html;charset=utf-8
Content-Length: 1061
Date: Fri, 12 Jul 2013 14:04:25 GMT
```

If this request were sent by the browser, the authentication challenge would prompt the user for credentials using a simple user/password dialog.

Let's now **provide the correct credentials** and send the request again:

```
curl -i --digest --user
```

```
       user1:user1Pass       http://localhost/spring-security-mvc-digest-auth/
homepage.html
```

Notice that we are enabling Digest Authentication for the *curl* command via the *–digest* flag.

The first response from the server will be the same – the *401 Unauthorized* – but the challenge will now be interpreted and acted upon by a second request – which will succeed with a *200 OK*:

```
HTTP/1.1 401 Unauthorized
Server: Apache-Coyote/1.1
Set-Cookie:   JSESSIONID=A961E0D...;   Path=/spring-security-mvc-digest-auth/;
 HttpOnly
WWW-Authenticate:  Digest  realm="Contacts  Realm  via  Digest  Authentication",
 qop="auth",
 nonce="MTM3MzYzODgyOTczMTo3YjM4OWQzMGU0YTgwZDg0YmYwZjRlZWJjMDQzZWZkOA=="
Content-Type: text/html;charset=utf-8
Content-Length: 1061
Date: Fri, 12 Jul 2013 14:15:29 GMT

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie:   JSESSIONID=55F996B...;   Path=/spring-security-mvc-digest-auth/;
 HttpOnly
Content-Type: text/html;charset=ISO-8859-1
Content-Language: en-US
Content-Length: 90
Date: Fri, 12 Jul 2013 14:15:29 GMT

<html>
<head></head>

<body>
 <h1>This is the homepage</h1>
</body>
</html>
```

A final note on this interaction is that a client can **preemptively send the correct Authorization header** with the first request, and thus entirely **avoid the server security challenge** and the second request.

4. The Maven Dependencies

The security dependencies are discussed in depth in the Spring Security Maven tutorial [http://www.baeldung.com/spring-security-with-maven]. In short, we will need to define *spring-security-web* and *spring-security-config* as dependencies in our *pom.xml*.

5. Conclusion

In this tutorial we introduce security into a simple Spring MVC project by leveraging the Digest Authentication support in the framework.

The implementation of these examples can be found in the github project [https://github.com/eugenp/tutorials/tree/master/spring-security-mvc-digest-auth#readme] â€" this is an Eclipse based project, so it should be easy to import and run as it is.

When the project runs locally, the homepage html can be accessed at (or, with minimal Tomcat configuration, on port 80):

http://localhost:8080/spring-security-mvc-digest-auth/homepage.html

Finally, there is no reason an application needs to choose between Basic and Digest authentication [http://www.baeldung.com/2011/11/20/basic-and-digest-authentication-for-a-restful-service-with-spring-security-3-1/] – **both can be set up simultaneously on the same URI structure**, in such a way that the client can pick between the two mechanisms when consuming the web application.

http://twitter.com/share

security [http://www.baeldung.com/tag/security/], Spring [http://www.baeldung.com/tag/spring/]

Basic and Digest Authentication for a REST Service with Spring Security

Return to Content

Contents

If you're new here, you may want to get my "REST APIs with Spring" eBook [https://my.leadpages.net/leadbox/146382273f72a2%3A13a71ac76b46dc/5735865741475840/]. Thanks for visiting!

Table of Contents

1. Overview

This article discusses how to **set up both Basic and Digest Authentication on the same URI structure of a REST API**. In a previous article, we discussed another method of securing the REST Service – form based authentication [http://www.baeldung.com/2011/10/31/securing-a-restful-web-service-with-spring-security-3-1-part-3/], so Basic and Digest authentication is the natural alternative, as well as the more RESTful one.

2. Configuration of Basic Authentication

The main reason that form based authentication is not ideal for a RESTful Service is that Spring Security will **make use of Sessions –** this is of course state on the server, so **the statelessness constraints in REST** is practically ignored.

We'll start by setting up Basic Authentication – first we remove the old custom entry point and filter from the main *<http>* security element:

```
<http create-session="stateless">
 <intercept-url pattern="/api/admin/**" access="ROLE_ADMIN" />

 <http-basic />
</http>
```

Note how support for basic authentication has been added with a single configuration line – *<http-basic />* – which handles the creation and wiring of both the *BasicAuthenticationFilter* and the *BasicAuthenticationEntryPoint*.

2.1. Satisfying the stateless constraint – getting rid of sessions

One of the main constraints of the RESTful architectural style is that the client-server communication is fully **stateless**, as the original dissertation [http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm] reads:

> Â Â Â **5.1.3 Stateless**
>
> We next add a constraint to the client-server interaction: communication must be stateless in nature, as in the client-stateless-server (CSS) style of Section 3.4.3 (Figure 5-3), such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. **Session state is therefore kept entirely on the client**.

The concept of **Session** on the server is one with a long history in Spring Security, and removing it entirely has been difficult until now, especially when configuration was done by using the namespace. However, Spring Security 3.1 augments [https://jira.springsource.org/browse/SEC-1424] the namespace configuration with a **new stateless option** for session creation, which effectively guarantees that no session will be created or used by Spring. What this new option does is completely removes all session related filters from the security filter chain, ensuring that authentication is performed for each request.

3. Configuration of Digest Authentication

Starting with the previous configuration, the filter and entry point necessary to set up digest authentication will be defined as beans. Then, the **digest entry point** will override the one created by *<http-basic>* behind the scenes. Finally, the custom **digest filter** will be introduced in the security filter chain using the *after* semantics of the security namespace to position it directly after the basic authentication filter.

```
<http create-session="stateless" entry-point-ref="digestEntryPoint">
 <intercept-url pattern="/api/admin/**" access="ROLE_ADMIN" />

 <http-basic />
 <custom-filter ref="digestFilter" after="BASIC_AUTH_FILTER" />
</http>

<beans:bean id="digestFilter" class=
 "org.springframework.security.web.authentication.www.DigestAuthenticationFilter">
 <beans:property name="userDetailsService" ref="userService" />
 <beans:property name="authenticationEntryPoint" ref="digestEntryPoint" />
</beans:bean>

<beans:bean id="digestEntryPoint" class=
 "org.springframework.security.web.authentication.www.DigestAuthenticationEntryPoint">
   <beans:property   name="realmName"   value="Contacts   Realm   via   Digest
 Authentication"/>
 <beans:property name="key" value="acegi" />
</beans:bean>

<authentication-manager>
 <authentication-provider>
 <user-service id="userService">
 <user name="eparaschiv" password="eparaschiv" authorities="ROLE_ADMIN" />
 <user name="user" password="user" authorities="ROLE_USER" />
 </user-service>
 </authentication-provider>
</authentication-manager>
```

Unfortunately there is no support [https://jira.springsource.org/browse/SEC-1860] in the security namespace to automatically configure the digest authentication the way basic authentication can be configured with *<http-basic>*. Because of that, the necessary beans had to be defined and wired manually into the security configuration.

4. Supporting both authentication protocols in the same RESTful service

Basic or Digest authentication alone can be easily implemented in Spring Security 3.x; it is supporting both of them for the same RESTful web service, on the same URI mappings that introduces a new level of complexity into the configuration and testing of the service.

4.1. Anonymous request

With both basic and digest filters in the security chain, the way a **anonymous request** – a request containing no authentication credentials (*Authorization* HTTP header) – is processed by Spring Security is – the two authentication filters will find **no credentials** and will continue execution of the filter chain. Then, seeing how the request wasn't authenticated, an *AccessDeniedException* is thrown and caught in the *ExceptionTranslationFilter*, which commences the digest entry point, prompting the client for credentials.

The responsibilities of both the basic and digest filters are very narrow – they will continue to execute the security filter chain if they are unable to identify the type of authentication credentials in the request. It is because of this that Spring Security can have the flexibility to be configured with support for multiple authentication protocols on the same URI.

When a request is made containing the correct authentication credentials – either basic or digest – that protocol will be rightly used. However, for an anonymous request, the client will get prompted only for digest authentication credentials. This is because the digest entry point is configured as the main and single entry point of the Spring Security chain; as such **digest authentication can be considered the default**.

4.2. Request with authentication credentials

A **request with credentials** for Basic authentication will be identified by the *Authorization* header starting with the prefix *"Basic"*. When processing such a request, the credentials will be decoded in the basic authentication filter and the request will be authorized. Similarly, a request with credentials for Digest authentication will use the prefix *"Digest"*Â for it’s *Authorization* header.

5. Testing both scenarios

The tests will consume the REST service by creating a new resource after authenticating with either basic or digest:

```
@Test
public                                                                     void
 givenAuthenticatedByBasicAuth_whenAResourceIsCreated_then201IsReceived(){
 // Given
 // When
 Response response = given()
 .auth().preemptive().basic( ADMIN_USERNAME, ADMIN_PASSWORD )
 .contentType( HttpConstants.MIME_JSON ).body( new Foo( randomAlphabetic( 6 ) ) )
 .post( paths.getFooURL() );

 // Then
 assertThat( response.getStatusCode(), is( 201 ) );
}
@Test
public                                                                     void
 givenAuthenticatedByDigestAuth_whenAResourceIsCreated_then201IsReceived(){
 // Given
 // When
 Response response = given()
 .auth().digest( ADMIN_USERNAME, ADMIN_PASSWORD )
 .contentType( HttpConstants.MIME_JSON ).body( new Foo( randomAlphabetic( 6 ) ) )
 .post( paths.getFooURL() );

 // Then
 assertThat( response.getStatusCode(), is( 201 ) );
}
```

Note that the test using basic authentication adds credentials to the request **preemptively**, regardless if the server has challenged for authentication or not. This is to ensure that the server doesn’t need to challenge the client for credentials, because if it did, the challenge would be for Digest credentials, since that is the default.

6. Conclusion

This article covered the configuration and implementation of both Basic and Digest authentication for a RESTful service, using mostly Spring Security 3.0 namespace support as well as some new features added by Spring Security 3.1.

For the full implementation, check out the github project [https://github.com/eugenp/REST#readme].

http://twitter.com/share

REST [http://www.baeldung.com/tag/rest/], security [http://www.baeldung.com/tag/security/], Spring [http://www.baeldung.com/tag/spring/]