**Spring Security Authentication Provider**

by **[Eugen Paraschiv](https://plus.google.com/u/0/+EugenParaschiv?rel=author" \o "Visit Eugen Paraschiv’s website)** on June 2, 2013 in [**Security**](http://www.baeldung.com/category/security-2/), [**Spring**](http://www.baeldung.com/category/spring/)

If you're new here, [you may want to get my "Spring Development Report"](https://my.leadpages.net/leadbox/147a1bd73f72a2%3A13a71ac76b46dc/5767123439714304/). Thanks for visiting!

**I usually post about Security on Twitter - you can follow me there:**

[Follow @baeldung](https://twitter.com/baeldung)

**1. Overview**

This tutorial will show how to set up an **Authentication Provider in Spring Security** to allow for additional flexibility compared to the standard scenario using a simple*UserDetailsService*.

**2. The Authentication Provider**

Spring Security provides a variety of options for performing authentication – all following a simple contract – **an *Authentication* request is processed by an*AuthenticationProvider*** and a fully authenticated object with full credentials is returned.

The standard and most common implementation is the *DaoAuthenticationProvider* – which retrieves the user details from a a simple, read only user DAO – the *UserDetailsService*. This User Details Service **only has access to the username** in order to retrieve the full user entity – and in a large number of scenarios, this is enough.

More custom scenarios will still need to access the full *Authentication* request to be able to perform the authentication process – for example, when authenticating against some external, third party service (such as [Crowd](http://www.atlassian.com/software/crowd/overview)) – **both the *username* and the *password*from the authentication request will be necessary**.

For these, more advanced scenarios, we’ll need to **define a custom Authentication Provider**:

|  |
| --- |
| @Component  public class CustomAuthenticationProvider implements AuthenticationProvider {        @Override      public Authentication authenticate(Authentication authentication)        throws AuthenticationException {          String name = authentication.getName();          String password = authentication.getCredentials().toString();            // use the credentials to try to authenticate against the third party system          if (authenticatedAgainstThirdPartySystem()) {              List<GrantedAuthority> grantedAuths = new ArrayList<>();              return new UsernamePasswordAuthenticationToken(name, password, grantedAuths);          } else {              throw new AuthenticationException("Unable to auth against third party systems");          }      }        @Override      public boolean supports(Class<?> authentication) {          return authentication.equals(UsernamePasswordAuthenticationToken.class);      } |

Notice that the granted authorities set on the returned *Authentication* object are empty – this is because authorities are of course application specific.

**3. Register the Auth Provider**

Now that the Authentication Provider is defined, we need to specify it in the XML Security Configuration, using the available namespace support:

|  |
| --- |
| <?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-3.2.xsd">        <http use-expressions="true">          ...          <http-basic/>      </http>        <authentication-manager>          <authentication-provider ref="customAuthenticationProvider" />      </authentication-manager>    </beans:beans>      <authentication-manager>          <authentication-provider ref="customAuthenticationProvider" />      </authentication-manager>  </beans:beans> |

**4. Do Authentication**

Requesting Authentication from the Client is basically the same with or without this custom authentication provider on the back end – we can use a simple *curl* command to send an authenticated request:

|  |
| --- |
| curl --header "Accept:application/json" -i --user user1:user1Pass      http://localhost:8080/spring-security-custom/api/foo/1 |

Note that – for the purposes of this example – we have secured the REST API with Basic Authentication.

And we get back the expected 200 OK from the Server:

|  |
| --- |
| HTTP/1.1 200 OK  Server: Apache-Coyote/1.1  Set-Cookie: JSESSIONID=B8F0EFA81B78DE968088EBB9AFD85A60; Path=/spring-security-custom/; HttpOnly  Content-Type: application/json;charset=UTF-8  Transfer-Encoding: chunked  Date: Sun, 02 Jun 2013 17:50:40 GMT |

**5. Conclusion**

In this article we discussed an example of custom authentication provider for Spring Security.

[The implementation of this *Spring Security Authentication Provider* can be downloaded as a working sample project.](https://my.leadpages.net/leadbox/142a36073f72a2%3A13a71ac76b46dc/5748809699164160/)