Securing Wicket Applications with Java EE, Spring & Jetty

This paper describes how to secure Apache Wicket applications using Java EE container and Spring security inside of Jetty servlet container.

Java EE Security

The reasons for using Java EE security are well documented and will not be described here. This paper assumes you have already made your decision and describes how to enable form-based authentication and coarse-grained role enforcement.

Spring Security

Likewise Spring security has characteristics that won't be explained in this document. There are other documents in public domain that cover this topic. This paper will describe how to configure Spring security to coexist with Java EE security. The Spring enforcement used here is FilterSecurityInterceptor for page level security.

Configuring Java EE Security

Obviously this step is wholly dependent on which servlet container you are using. This how-to guide keeps things simple by showing you how to do this using embedded Jetty's file based security. While Jetty file security is convenient for this tutorial and development environments it is not suitable for production environments. For that you should use a non-file based security provider like Fortress: http://iamfortress.org

Step 1: Create Jetty User Properties File

This example has three users: user1, user2 and user3. Each will have a password of 'password'. It also defines three roles: ROLE_TEST1, ROLE_TEST2, ROLE_TEST3. For our example, User1 will be assigned all three roles. User's 2 and 3 will receive ROLE_TEST2 and ROLE_TEST3 respectively.

Note: Each role has a prefix of 'ROLE_'. This convention is due to Spring's preference for role names. It is possible to override this Spring behavior and drop the prefix but this topic will not be covered here.

Create a new file called jetty-user.properties and add the following lines:

```
user1: password, ROLE_TEST1,ROLE_TEST2,ROLE_TEST3

user2: password, ROLE_TEST2

user3: password, ROLE_TEST3
```

Step 2: Configure Jetty to use the Create Jetty User Properties File

Edit the jetty startup class and add the following lines. Make sure the config points to the location you saved your properties file on previous step.

```
// Setup the test security realm, its name must match what's in the
web.xml's 'realm-name' tag:

HashLoginService dummyLoginService = new
HashLoginService("MySecurityRealm");

dummyLoginService.setConfig("src/test/resources/jetty-
users.properties");

bb.getSecurityHandler().setLoginService( dummyLoginService );

server.setHandler(bb);
```

Step 3: Create a new folder to store the Form-based artifacts

Under the webapp folder in your application add a new folder called 'login'. This will be used to store the Java EE login and error forms needed for Java EE form-based security. The location of the folder is flexible but must be public.

Step 4: Create Java EE Login Form

Form based security can use JSP or HTML artifacts. For this example we'll simply use static HTML. What is important is the form must be placed in a location where it is not controlled by Wicket runtime. The login forms job is to collect the userid and password and map to predefined attribute names of ***j_username*** and ***j_password*** before submitting to ***j_security_check*** servlet that resides inside all compliant Java servlet container. The location of the form must be same as specified in Step 3. The style and format of the form are left for you.

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <h3> My Login Form </h3>
</head>
<body>
<FORM METHOD=POST ACTION="j_security_check">
    <fieldset>
        <legend>Login Operations</legend>
        <table id="inputFormTable">
            <tr>
                <td>
                    <label for="userId">UserID</label>
                </td>
                <td>
                    <input name="j_username" id="userId" type="text" class="formLarge"
style="width: 250px"/>
                </td>
            </tr>
            <tr>
                <td>
                    <label for="pswdField">Password</label>
                </td>
                <td>
                    <input id="pswdField" name="j_password" type="password" style="width:
250px"/>
                </td>
            </tr>
```

```
            <tr>
                <td>
                    <input type="submit" name="login" value="Login">
                </td>
            </tr>
        </table>
    </fieldset>
</form>
</body>
</html>
```

Step 5: Create Java EE Error Form

This form is used to display error message if the user did not enter their credentials correctly. There are two main concerns here:

Do not give away too much information to the user about what went wrong. From a security standpoint it is best to inform user the authentication failed without telling them why it failed. The second concern is to make sure the post action refers the user to the home page for you application.

```
<!DOCTYPE HTML PUBLIC "-//W3C/DTD HTML 4.0 Transitional//EN">
<html>
<head>
    <title> My Web Application </title>
</head>
<body>
<h3>Invalid userid or password detected</h3>
<FORM METHOD=POST ACTION="/mywebcontext/wicket/bookmarkable/com.myapp.LaunchPage">
    <p>
        <font size="2">Click the button to re-authenticate.</font>
        <BR><BR>
        <input type="submit" name="relogin" value="return">
    </p>
</form>
</body>
</html>
```

Step 6: Enable Java EE security

Edit the web.xml and add the following. The url-pattern tag includes the Wicket pages. The realm-name must map to the HashLoginService in Step 2. The form-login-page and form-error-page tags must map to the name and location of forms created in Steps 5 & 6.

```
<web-app>
...
        <!-- Begin JAVA EE Security configs: -->
        <security-constraint>
                <display-name>Commander Security Constraint</display-name>
                <web-resource-collection>
                        <web-resource-name>Protected Area</web-resource-name>
                        <!-- Define the context-relative URL(s) to be protected -->
                        <url-pattern>/wicket/*</url-pattern>
                </web-resource-collection>
                <auth-constraint>
                        <!-- Anyone with one of the listed roles may access this area -->
                        <role-name>ROLE_TEST1</role-name>
                        <role-name>ROLE_TEST2</role-name>
                        <role-name>ROLE_TEST3</role-name>
                </auth-constraint>
```

```
        </security-constraint>

        <login-config>
                <auth-method>FORM</auth-method>
                <realm-name>MySecurityRealm</realm-name>
                <form-login-config>
                        basedir
                        <form-login-page>/login/login.html</form-login-page>
                        <form-error-page>/login/error.html</form-error-page>
                </form-login-config>
        </login-config>

        <!-- Security roles referenced by this web application -->
        <security-role>
                <role-name>ROLE_TEST1</role-name>
        </security-role>
        <security-role>
                <role-name>ROLE_TEST2</role-name>
        </security-role>
        <security-role>
                <role-name>ROLE_TEST3</role-name>
        </security-role>
...
</web-app>
```

Step 6: Enable Spring security in Web.xml

If your application isn't already using Spring security you have to enable the Spring filter chain in the web.mxl and map it to a context file:

```
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:applicationContext.xml</param-value>
    </context-param>
    <listener>
        <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
…
    <filter>
        <filter-name>filterChainProxy</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>
    <filter-mapping>
        <filter-name>filterChainProxy</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
```

Step 7: Enable Spring security in applicationContext.xml

Most of the contents of this file are boilerplate and will not change from one implementation to the next. The items that are variable are included below. For this example we are using Spring Security 3.x.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:sec="http://www.springframework.org/schema/security"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
```

```
                            http://www.springframework.org/schema/beans/spring-beans.xsd
                            http://www.springframework.org/schema/beans
                            http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                            http://www.springframework.org/schema/util
                            http://www.springframework.org/schema/util/spring-util-3.0.xsd
                            http://www.springframework.org/schema/security
                            http://www.springframework.org/schema/security/spring-security.xsd">
    <!-- setup spring security using preauthenticated (java ee) strategy -->
    <bean id="filterChainProxy" class="org.springframework.security.web.FilterChainProxy">
        <sec:filter-chain-map path-type="ant">
            <sec:filter-chain pattern="/**"
filters="sif,j2eePreAuthFilter,logoutFilter,etf,fsi"/>
        </sec:filter-chain-map>
    </bean>
    <bean id="sif"
class="org.springframework.security.web.context.SecurityContextPersistenceFilter"/>
    <sec:authentication-manager alias="authenticationManager">
        <sec:authentication-provider ref='preAuthenticatedAuthenticationProvider'/>
    </sec:authentication-manager>
    <bean id="preAuthenticatedAuthenticationProvider"

class="org.springframework.security.web.authentication.preauth.PreAuthenticatedAuthenticationProv
ider">
        <property name="preAuthenticatedUserDetailsService"
ref="preAuthenticatedUserDetailsService"/>
    </bean>
    <bean id="preAuthenticatedUserDetailsService"

class="org.springframework.security.web.authentication.preauth.PreAuthenticatedGrantedAuthorities
UserDetailsService"/>
    <bean id="j2eePreAuthFilter"

class="org.springframework.security.web.authentication.preauth.j2ee.J2eePreAuthenticatedProcessin
gFilter">
        <property name="authenticationManager" ref="authenticationManager"/>
        <property name="authenticationDetailsSource">
            <bean
class="org.springframework.security.web.authentication.preauth.j2ee.J2eeBasedPreAuthenticatedWebA
uthenticationDetailsSource">
                <property name="mappableRolesRetriever">
                    <bean
class="org.springframework.security.web.authentication.preauth.j2ee.WebXmlMappableAttributesRetri
ever"/>
                </property>
                <property name="userRoles2GrantedAuthoritiesMapper">
                    <bean
class="org.springframework.security.core.authority.mapping.SimpleAttributes2GrantedAuthoritiesMap
per">
                        <property name="convertAttributeToUpperCase" value="true"/>
                    </bean>
                </property>
            </bean>
        </property>
    </bean>
    <bean id="preAuthenticatedProcessingFilterEntryPoint"
          class="org.springframework.security.web.authentication.Http403ForbiddenEntryPoint"/>
    <bean id="logoutFilter"
class="org.springframework.security.web.authentication.logout.LogoutFilter">
        <constructor-arg value="/"/>
        <constructor-arg>
            <list>
```

```
                <bean
class="org.springframework.security.web.authentication.logout.SecurityContextLogoutHandler"/>
            </list>
        </constructor-arg>
    </bean>
    <bean id="servletContext"
class="org.springframework.web.context.support.ServletContextFactoryBean"/>
    <bean id="etf" class="org.springframework.security.web.access.ExceptionTranslationFilter">
        <property name="authenticationEntryPoint"
ref="preAuthenticatedProcessingFilterEntryPoint"/>
    </bean>
    <bean id="httpRequestAccessDecisionManager"
class="org.springframework.security.access.vote.AffirmativeBased">
        <property name="allowIfAllAbstainDecisions" value="false"/>
        <property name="decisionVoters">
            <list>
                <ref bean="roleVoter"/>
            </list>
        </property>
    </bean>
    <bean id="fsi"
class="org.springframework.security.web.access.intercept.FilterSecurityInterceptor">
        <property name="authenticationManager" ref="authenticationManager"/>
        <property name="accessDecisionManager" ref="httpRequestAccessDecisionManager"/>
        <property name="securityMetadataSource">
            <sec:filter-invocation-definition-source>
                <!-- before spring interceptor recognizes these roles, the j2ee preauthentication
filter requires prior declaration in web.xml -->
                <sec:intercept-url pattern="/wicket/bookmarkable/com.myapp.page1"
                                   access="ROLE_TEST1"/>
                <sec:intercept-url pattern="/wicket/bookmarkable/com.myapp.page1"
                                   access="ROLE_TEST2"/>
                <sec:intercept-url pattern="/wicket/bookmarkable/com.myapp.page1"
                                   access="ROLE_TEST3"/>
            </sec:filter-invocation-definition-source>
        </property>
    </bean>
    <bean id="roleVoter" class="org.springframework.security.access.vote.RoleVoter"/>
    <bean id="securityContextHolderAwareRequestFilter"

class="org.springframework.security.web.servletapi.SecurityContextHolderAwareRequestFilter"/>
</beans>
```

Step 8: Add Spring security dependencies to your Wicket application.

This example uses Maven for dependency management. Here are the dependencies you will need:

```
         <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-web</artifactId>
            <version>3.2.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-expression</artifactId>
            <version>3.2.3.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.security</groupId>
            <artifactId>spring-security-core</artifactId>
            <version>$3.1.4.RELEASE</version>
```

```
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.1.4.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.1.4.RELEASE</version>
</dependency>
```

Step 9: Dealing with Session Timeout Problem

If you are using AJAX you will experience problems when the Java container times out the session after a period of inactivity of the user. In this scenario the user will not be notified of the timeout and the application appears to hang in the browser.

One workaround that can be used to deal with this problem is to add the following override to your AJAX buttons that redirect user to the home page. This will force the container to do the right thing and force the user to login again:

```
@Override
protected void updateAjaxAttributes( AjaxRequestAttributes attributes )
{
    super.updateAjaxAttributes( attributes );
    AjaxCallListener ajaxCallListener = new AjaxCallListener()
    {
        @Override
        public CharSequence getFailureHandler( Component component )
        {
            return "window.location.replace(\"/commander/home.html\");";
        }
    };
    attributes.getAjaxCallListeners().add( ajaxCallListener );
}
```