

Spring Acegi Tutorial

A tutorial on using the Spring Acegi Security Framework to secure web applications.



Bart van Riel

Capgemini Netherlands - Sector Public - Practice D75 "MOJO"

Version 1.1 - May 2006

Table of Contents

1	Introduction.....	3
1.1	Why this tutorial.....	3
1.2	Tutorial objectives.....	3
2	Sources, IDE & other bare necessities.....	4
2.1	Sources.....	4
2.2	IDE & Application server.....	4
2.3	Other bare necessities: Spring and Acegi.....	4
3	A short discussion on security.....	5
3.1	Authentication.....	5
3.2	Authorization.....	5
3.3	The Four Checks.....	6
4	The example application.....	7
4.1	Functionality.....	7
4.1.1	The Normal User area.....	7
4.1.2	The Administrator area.....	9
5	Acegi Web Security.....	11
5.1	The Authentication object	11
5.2	Filters.....	11
5.3	Configuration.....	11
5.3.1	The Filter Chain.....	12
5.3.2	The AuthenticationProcessingFilter.....	12
5.3.3	The HttpSessionContextIntegrationFilter.....	14
5.3.4	The ExceptionTranslationFilter.....	14
5.3.5	FilterSecurityInterceptor.....	15
5.4	Using an authentication database through JDBC.....	18
6	To conclude.....	19
7	Resources.....	19

1 Introduction

This tutorial describes the configuration of webapplication security using the Acegi Security Framework for Spring.

1.1 *Why this tutorial*

Configuration of Acegi is a complex task Acegi has a rich architecture for implementing security and many options to configure it. Although there are numerous tutorials and book chapters devoted to this, I have had to consult several documentation sources and combine the provided information to get the complete picture. All kinds of problems arose (outdated configuration examples, mixing current and deprecated versions of the various frameworks leading to strange configuration exceptions and so on and so on). Apparently, implementing webapplication security using Acegi is powerful but not a very trivial task. I hope to give the interested reader a compact overview of “the way to do it”, simply making things work without going into details of the frameworks and without the sidestepping into additional (but non-essential) configuration options most books delve into.

1.2 *Tutorial objectives*

This tutorial should give the reader:

- An understanding of the basic principles and the mechanics of webapplication security;
- Insight in the mechanics of Acegi to achieve this security;
- An understanding of the configuration options to make Acegi do it's thing;
- Providing a sample application for the developer to build upon for yer' own fun & games.

2 Sources, IDE & other bare necessities

2.1 Sources

The sources for this tutorial can be found in “SpringAcegiTutorial.zip”. It is an Eclipse WebTools Project web project and should be imported into the Eclipse WTP workspace.

2.2 IDE & Application server

I used Eclipse WebTools Project (WTP) 1.0.2. It can be download from <http://download.eclipse.org/webtools/downloads/>. Look for the “all in one” bundle which gives you the entire IDE you need. Note that you need a fairly recent JDK to run it. I used Apache Tomcat 5.5 as my application server. Get it through <http://tomcat.apache.org>. Getting the Windows installer .exe is the easiest option (provided you run Windows, of course). Note that you will need J2SE 5.0 to run this release of tomcat.

2.3 Other bare necessities: Spring and Acegi

I used Spring version 1.2-rc1. You may download it from <http://www.springframework.org>, but the required libraries have already been included in the “SpringAcegiTutorial.zip” file (to be honest, I have included too much because I packaged the complete spring.jar in stead of the selected spring-xxx.jar's but hey, I am the stereotypical lazy programmer).

For Acegi I included the Acegi library jar in the zip file, which is version 1.0.0-RC2. You can download the full stuff from <http://www.acegisecurity.org> if you wish, but it is not necessary for this tutorial.

3 A short discussion on security

Before we delve into any code and Acegi's configuration, a few words on security. In particular, a few words on authentication, authorization and the steps you go through when requesting a resource from a secure webapplication.

3.1 Authentication

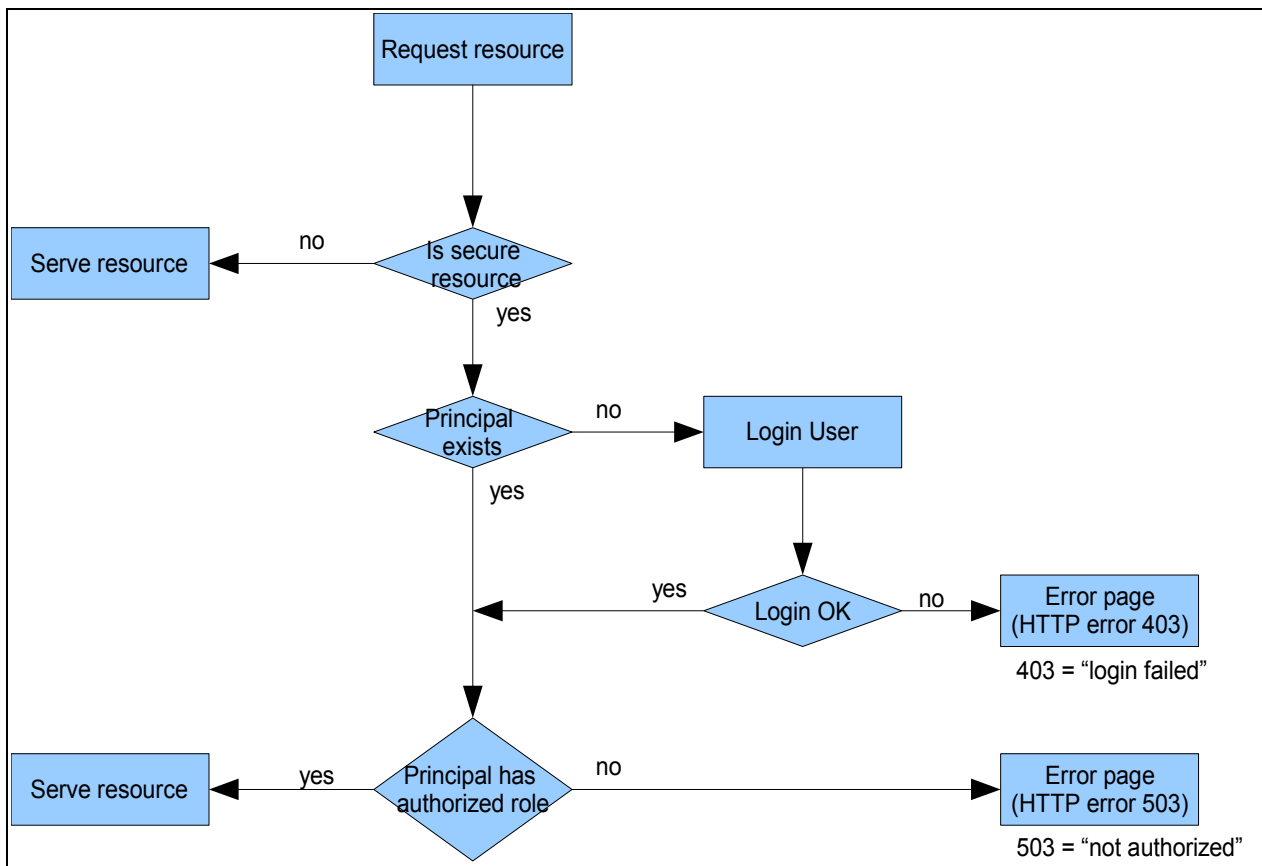
Authentication pertains to the question "Who are you?". Usually a user authenticates himself by successfully associating his "principal" (often a username) with his "credentials" (often a password).

3.2 Authorization

Authorization pertains to the question "What may you do?". In J2EE applications, this is achieved by making secured resources accessible ("requestable" in web applications) to particular "roles". Principals (i.e. users) who are associated with one or more of these roles will have access to those resources.

The motions of a secured web application

So what happens when you access a secured web application resource? The diagram below shows the typical rundown of accessing a web resource with security enabled.



And now in verbose mode: the usual path is 1) check if the resource is secured; 2) check if the requesting user has been authenticated; 3) check if the authenticated user is properly authorized to access the requested resource and 4) serve the requested resource. If the user has not been authenticated yet, walk through the Login dialog. If anything is out of order, display the corresponding error page. Or, if the resource is not secure, skip all previously mentioned steps and serve the resource right away.

3.3 The Four Checks

To make a long story short, security is implemented by these Four Checks:

1. the **Restricted Access** Check (is the resource secured?);
2. the **Existing Authentication** Check (has the user been authenticated?);
3. if there is no valid login for the user: the **Authentication Request** Check (are the correct username and password provided?);
4. the **Authorization** Check (does the user have the required roles?);

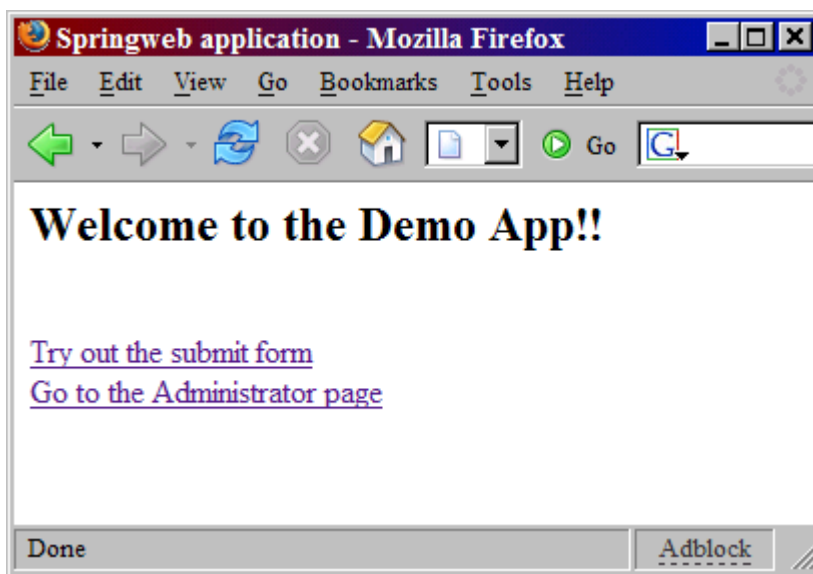
4 The example application

4.1 Functionality

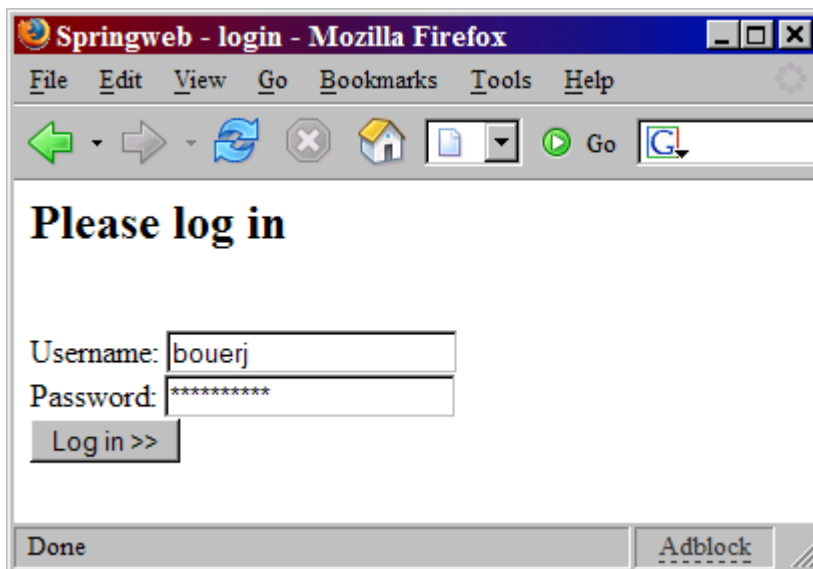
Since this tutorial is intended to give an example of Sprint Acegi configuration for web application authentication and authorization, that is exactly what the example application illustrates. The application has two “area's”, one area is only accessible to normal logged-in users and the other area is the administrator's page. Normal users and administrators do not have access to each other's area, only to their own. Of course, to access either of those area's you need to log in.

4.1.1 The Normal User area

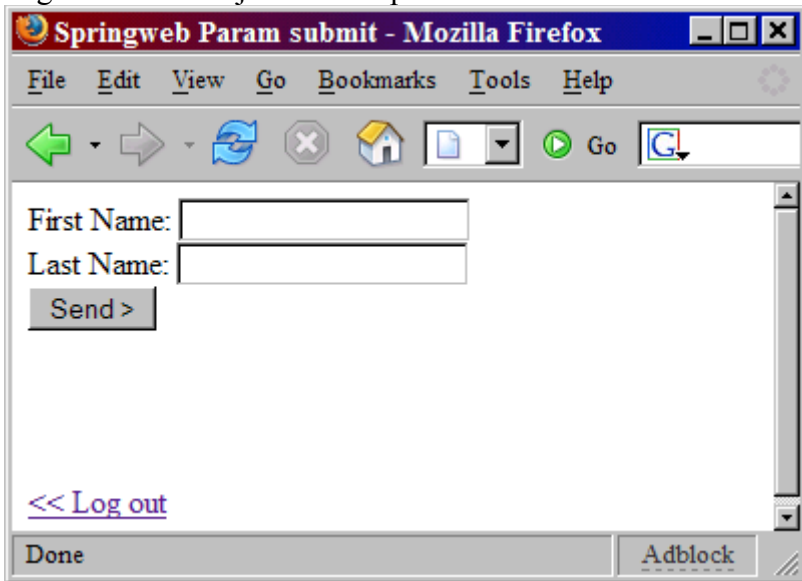
Here is the index page, accessible through “http://localhost:8080/springweb”:



Click on the “Try out the submit form” link. You should see the login page:

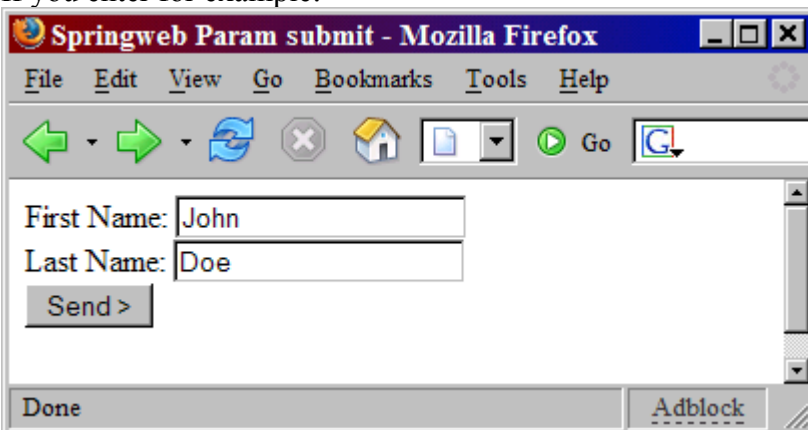


Login with “bouerj / ineedsleep”. You will be directed to the Submit Form:

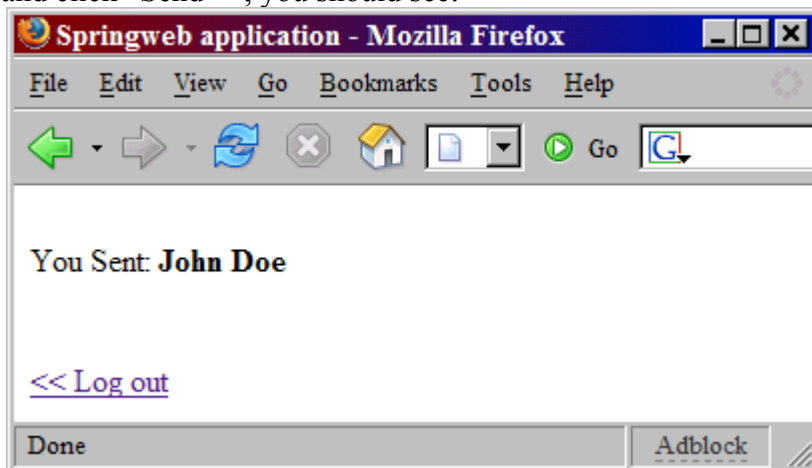


The “submit form” is the function for normal users and enables you to perform the incredibly useful task of submitting two text parameters which will be shown concatenated on the resulting page. There is also a validator on those two fields which prevents you from entering only spaces or nothing at all.

If you enter for example:



and click “Send >”, you should see:

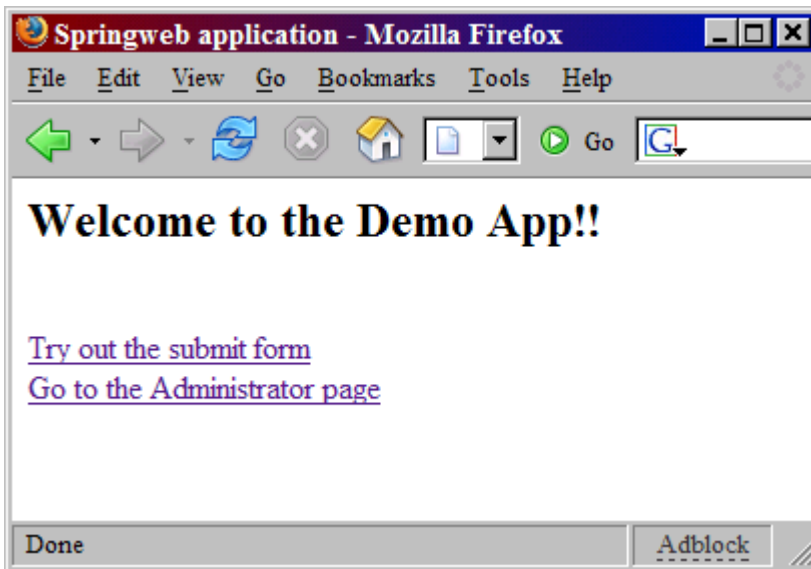


Advanced, is it not?

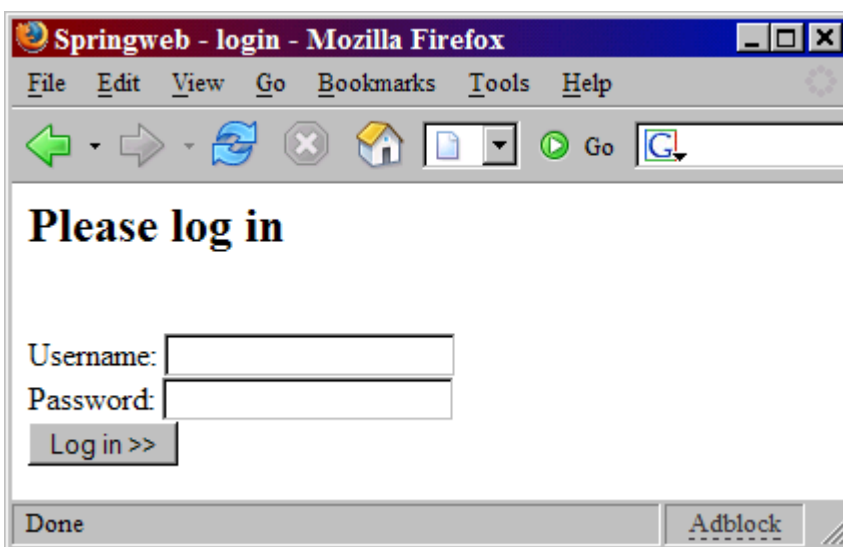
You can log off by clicking the “<< Log out” link, which calls the URL “<http://localhost:8080/springweb/logout.htm>”, which will take you back to the index page.

4.1.2 The Administrator area

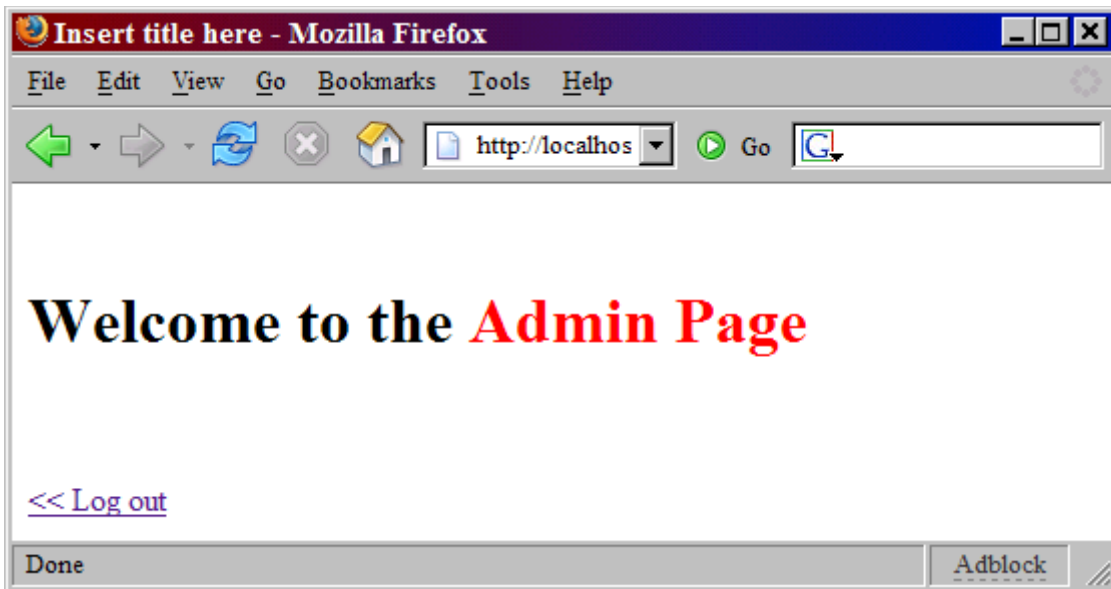
Open the browser again and fire up the index page:



Click on the “Go to the Administrator page” link and you should see the login page again:



Enter “jklaassen / 4moreyears” and you get to the “Administrator area”:



That's all!

So, you can start playing around a bit with various logins and see what happens when you use the wrong login for accessing an area. When you are ready (or fed up with it ;-) , read on to the part about configuring all this.

5 Acegi Web Security

5.1 The Authentication object

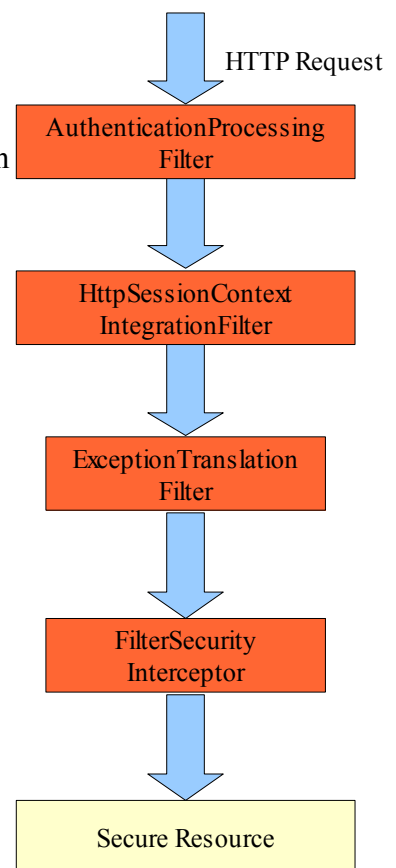
The Authentication object is pivotal to the Acegi framework. Since “security” basically means “restricted access for specific roles” the framework has to be able to determine, at any time, the roles given to the authenticated user. The framework stores this info in the “Authentication” object, which contains the username, password and the roles granted to the user.

The Authentication object is created and validated by the by the AuthenticationManager. Access to resources is controlled by the AccessDecisionManager.

5.2 Filters

Acegi uses a chain of (at least) three filters to enable webapplication security:

1. The AuthenticationProcessingFilter handles the Authentication Request Check (“logging into the application”). It uses the AuthenticationManager to do its work.
2. The HttpSessionContextIntegrationFilter maintains the Authentication object between various requests and passes it around to the AuthenticationManager and the AccessDecisionManager when needed;
3. The ExceptonTranslationFilter performs the Existing Authentication Check, handles security exceptions and takes the appropriate action. This action can be either spawning the authentication dialog (a.k.a. the login form) or returning the appropriate HTTP security error code. ExceptonTranslationFilter depends on the next filter, FilterSecurityInterceptor, to do its work.
4. FilterSecurityInterceptor manages the Restricted Acces Check,and the Authorisation check. It knows which resources are secure and which roles have access to them. FilterSecurityInterceptor uses the AuthenticationManager and AccessDecisionManager to do its work.



In good Spring and Dependency Injection fashion, the classes described above do not do their work alone and serve mainly as proxies who delegate the hard work to other classes. I will describe those classes in more detail while laying out the configuration file later on.

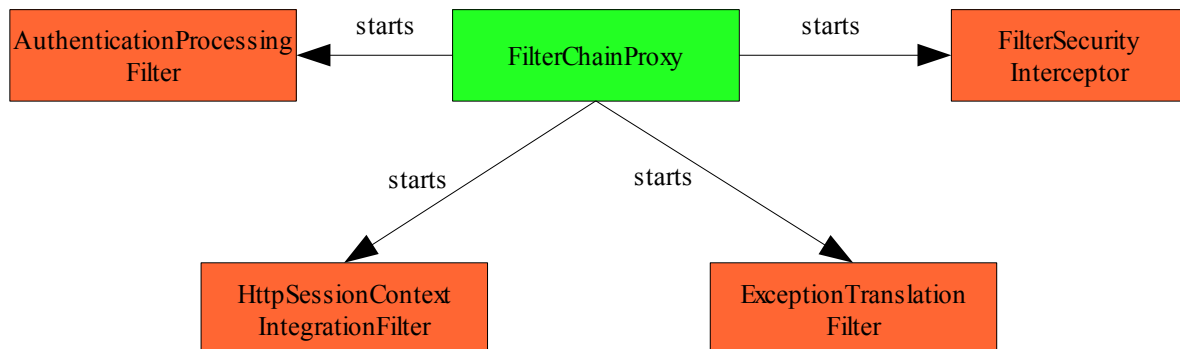
5.3 Configuration

Since Acegi depends on the Spring framework, all configuration is done through “wiring”. Without going into too much detail, “wiring” means associating JavaBeans with each other via a XML

configuration file. The three filters mentioned in the previous paragraph need all their dependent objects “wired” into them. We will get to the XML configuration in a moment, but first it might be a good idea to graphically outline the dependencies between the three filters and the various “utility objects” they use.

5.3.1 The Filter Chain

As you saw in the graph in paragraph 5.2, the three filters form a chain through which every HTTP request passes. Together, the three filters perform the task of securing the application. The three filters are “chained” together by an object called the “filterChainProxy”, which in turn creates and starts the three filters. See the diagram below:



In good Spring fashion, the three filters are started by yet another object, the “FilterChainProxy”. This proxy is configured in the configuration XML file and any additional filters (we will come to some of them later on) will be added to the “FilterChainProxy” configuration list.

The XML configuration is like this:

```
<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /**=httpSessionContextIntegrationFilter,formAuthenticationProcessingFilter,
        exceptionTranslationFilter,filterSecurityInterceptor
    </value>
  </property>
</bean>
```

The above configuration states the filter beans which will be started by the proxy. The configuration of those filter beans will be discussed below.

5.3.2 The AuthenticationProcessingFilter

The first filter through which the HTTP request will pass is the formAuthenticationProcessingFilter bean. This filter specializes in handling authentication request, i.e. The validation of username/password combinations. Let's take a look at the configuration XML:

```

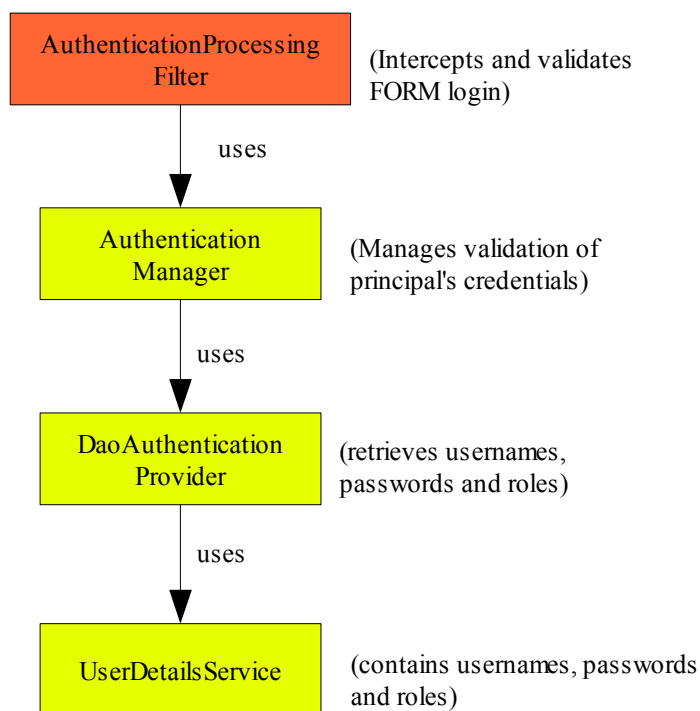
<bean id="formAuthenticationProcessingFilter"
      class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">
  <property name="filterProcessesUrl">
    <value>/j_acegi_security_check</value>
  </property>
  <property name="authenticationFailureUrl">
    <value>/loginFailed.html</value>
  </property>
  <property name="defaultTargetUrl">
    <value>/</value>
  </property>
  <property name="authenticationManager">
    <ref bean="authenticationManager" />
  </property>
</bean>

```

The filter bean is of type `org.acegisecurity.ui.webapp.AuthenticationProcessingFilter`. This filter class is specifically used for Form logins, which is why the form-submit URL (“filterProcessUrl”), the login-failed page (“authenticationFailureUrl”) are configured with this bean.

In case you are wondering where the login page itself is configured: with the security realm, which we will get to later on. Remember that the `AuthenticationProcessingFilter` specialised in *handling* authentication requests. Spawning a login dialog is enables a user to log in, but has nothing to do with actually validating the provided username/password combination and is therefore not configured in this filter.

For clarity, here's a diagram of `AuthenticationProcessingFilter` and its dependencies:

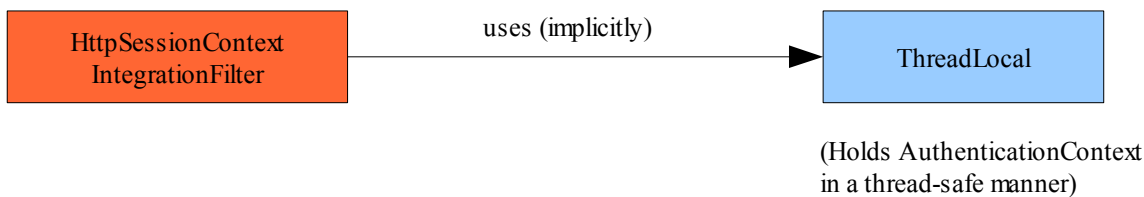


5.3.3 The HttpSessionContextIntegrationFilter

The work of the HttpSessionContextIntegrationFilter is very specialized and therefore very easy to configure. The only thing this filter does, is propagating the established authentication object through all requests. The filter wraps the authentication object a ThreadLocal and hands that wrapper over to the other filters in the chain. Here is the configuration XML:

```
<bean id="httpSessionContextIntegrationFilter"
      class="org.acegisecurity.context.HttpSessionContextIntegrationFilter">
</bean>
```

Below is a (perhaps unnecessary?) diagram of HttpSessionContextIntegrationFilter and its dependencies:



5.3.4 The ExceptionTranslationFilter

The ExceptionTranslationFilter is the one of the two “pivotal” filters in the security system (the other being FilterSecurityInterceptor). In short, ExceptionTranslationFilter catches any authentication or authorization error (in the form of an AcegiSecurityException) and may do one of the following two things.

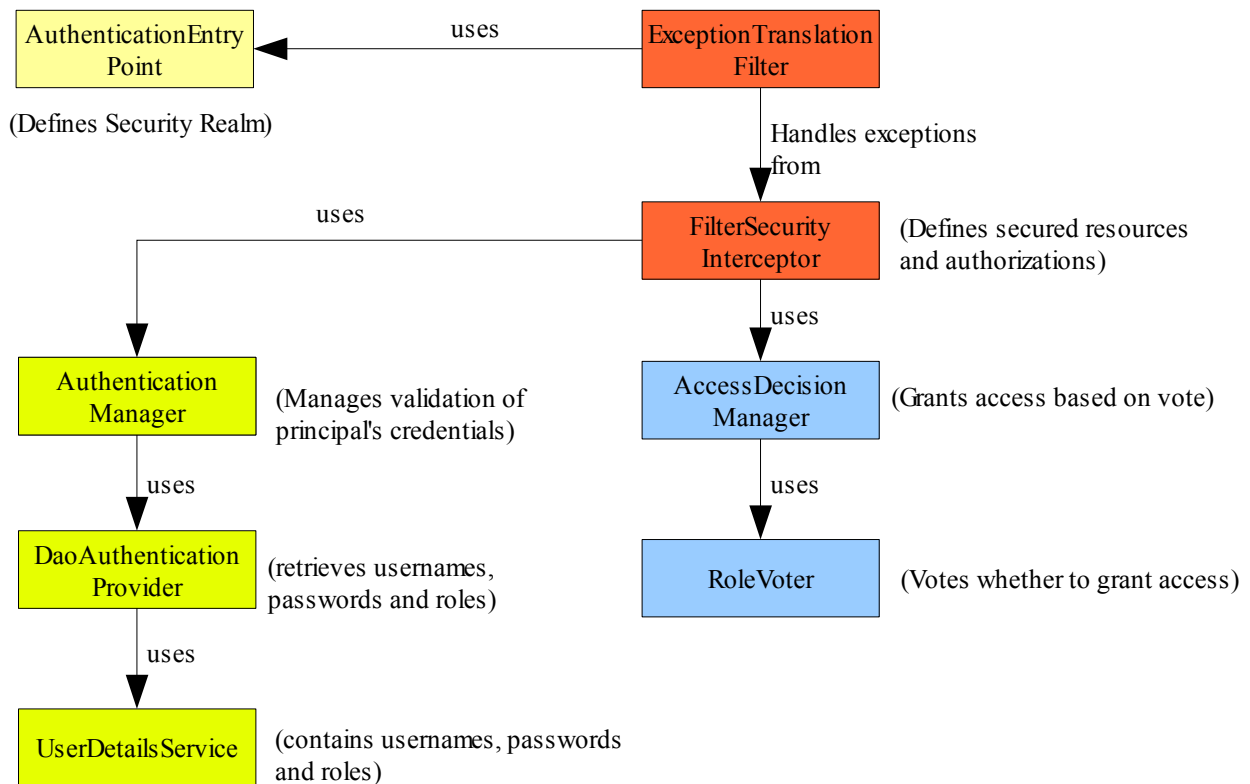
If the exception was caused by the absence of an Authentication object (i.e. the user has not logged in yet), it spawns the configured AuthenticationEntryPoint to prompt the user for login (more on AuthenticationEntryPoint later).

If the exception was caused by an authorization exception thrown by FilterSecurityInterceptor (i.e. the user is logged in but is not authorized for the resource requested), ExceptionTranslationFilter will send an SC_FORBIDDEN (HTTP 403) error to the browser, which will display its built-in version of an ‘unauthorized access’ page.

That sounds like quite the story, does it not? Nonetheless, the XML configuration for ExceptionTranslationFilter is rather simple:

```
<bean id="exceptionTranslationFilter"
      class="org.acegisecurity.ui.ExceptionTranslationFilter">
  <property name="authenticationEntryPoint">
    <ref bean="formLoginAuthenticationEntryPoint" />
  </property>
</bean>
```

The filter leaves all the hard work to its collaborators: FilterSecurityInterceptor (to which it is linked through the filter chain) and authenticationEntryPoint. Before we examine those two in more detail, it will definitely be useful to take a look at the following diagram, which shows the two filters and their dependencies:



5.3.5 FilterSecurityInterceptor

FilterSecurityInterceptor contains the definitions of the secured resources. Let's take a look at the XML configuration first:

```

<bean id="filterSecurityInterceptor"
  class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager">
    <ref bean="authenticationManager" />
  </property>
  <property name="accessDecisionManager">
    <ref bean="accessDecisionManager" />
  </property>
  <property name="objectDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /secure/admin/*=ROLE_ADMIN
      /secure/app/*=ROLE_USER
    </value>
  </property>
</bean>

```

You see two bean references here, “authenticationManager” and “accessDecisionManager”. We will get to those in a moment, first let's take a look at the property “objectDefinitionSource”. In Acegi security, “secured resources” are called “object definitions” (it is a rather generic sounding name because Acegi can be also used to control access to method invocations and object creations, not just web applications). The thing to remember here is that “objectDefinitionSource” should contain some directives and the URL patterns to be secured, along with the roles who have access to

those URL patterns.

There are two directives here:

- `CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON` tells the Acegi framework to convert the request URL to lowercase, so that the security mechanism cannot be surpassed by simply entering a slightly different URL in the browser;
- `PATTERN_TYPE_APACHE_ANT` makes it easier to define the URL patterns to be secured (there is also a different notation style, which looks more like regular expressions and less like the standard J2EE notation so I will not go into that here, mainly because I do not really understand that notation myself ;-)).

And there are two URL patterns defined here, which are the two main URLs used in the example webapplication.

AuthenticationManager

Here is the configuration XML for AuthenticationManager and it's dependents:

```
<bean id="authenticationManager"
      class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref bean="daoAuthenticationProvider" />
    </list>
  </property>
</bean>

<bean id="daoAuthenticationProvider"
      class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="authenticationDao">
    <ref bean="authenticationDao" />
  </property>
</bean>

<bean id="authenticationDao"
      class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      jklaassen=4moreyears,ROLE_ADMIN
      bouerj=inneedsleep,ROLE_USER
    </value>
  </property>
</bean>
```

The AuthenticationManager is of type ProviderManager, which means that it forms a proxy to the AuthenticationProvider. In Acegi, an AuthenticationProvider validates the inputted username/password combination and extracts the role(s) appointed to that user.

AuthenticationProvider is itself a proxy to an AuthenticationDao, which is basically an registry containing usernames, passwords and roles. There are several types of AuthenticationDao (in-memory, database via JDBC or even LDAP), but for simplicity the standard InMemoryDaoImpl type is used. In the Dao, two users have been defined (jklaassen and bouerj), each with a different role.

AccessDecisionManager

Validating the correctness of the username/password combination and the retrieval of associated roles is one thing, deciding whether to grant access is another. In other words: once a user has been

authenticated, he must also be *authorized*. This decision is the responsibility of the `AccessDecisionManager`. The `AccessDecisionManager` takes the available user information and decides to grant access (or not, of course). The `AccessDecisionManager` uses a `Voter` to determine if the user will be authorized. Below is the configuration XML:

```
<bean id="accessDecisionManager"
      class="org.acegisecurity.vote.UnanimousBased">
  <property name="decisionVoters">
    <list>
      <ref bean="roleVoter" />
    </list>
  </property>
</bean>

<bean id="roleVoter" class="org.acegisecurity.vote.RoleVoter">
  <property name="rolePrefix">
    <value>ROLE_</value>
  </property>
</bean>
```

The above configuration is usually sufficient for webapplications, so we will leave it at that. Note though, that you will have to specify which rolenames should be handled by a specific voter by specifying the role prefix.

Note also that it is possible to wire multiple voters into one `AccessDecisionManager` and that multiple `ProviderManagers` can be wired to an `AuthenticationManager`. So it is possible to let Acegi consult several different username/password registries available (say a mixture of LDAP, Database and NT Domain registries), with many different rolenames configured and voted on by several `Voters`. I admit that elaborate scenario to be a bit far fetched and certainly far too complex for one webapplication, but in huge enterprise systems such (legacy) complexity may very well exist.

AuthenticationEntryPoint

Now only one configuration item needs to be specified: the `AuthenticationEntryPoint`, which is the starting point of the authentication dialog. If the `FilterSecurityInterceptor` determines that there is no available authentication object present, the `SecurityEnforcementFilter` will pass control to the `AuthenticationEntryPoint`. Here is the configuration XML:

```
<bean id="formLoginAuthenticationEntryPoint"
      class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilterEntryPoint">
  <property name="loginFormUrl">
    <value>/login.jsp</value>
  </property>
  <property name="forceHttps">
    <value>>false</value>
  </property>
</bean>
```

The `AuthenticationEntryPoint` in this example is of type `AuthenticationProcessingFilterEntryPoint`, which is specifically suitable for FORM login dialogs. Configuration is fairly straightforward, only the URL of the login form to spawn needs to be specified. An optional parameter “forceHttps”, may be set to “true” if you would like the username/password data to be sent as encrypted data.

5.4 Using an authentication database through JDBC

The above example uses the `InMemoryDaoImpl` as the `AuthenticationDAO`, to store usernames, passwords and roles. That's fine for simple testing purposes, but in the real world a user registry is usually a database. So here's some configuration for using the `JdbcDaoImpl` class of Acegi:

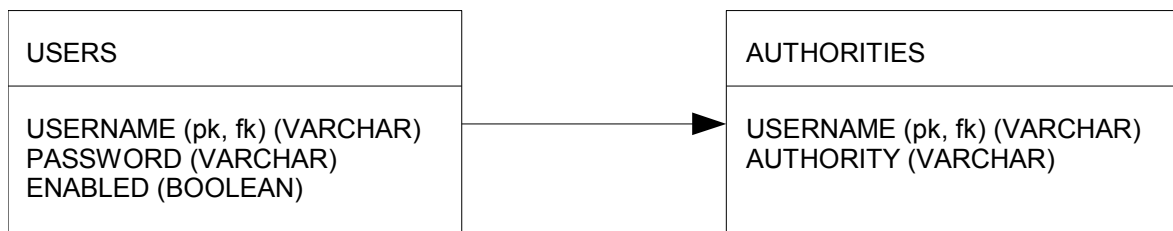
```
<bean id="userDetailsService"
      class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">
  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
</bean>
```

That's all. The `dataSource` configuration is standard Spring:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName">
    <value>com.mysql.jdbc.Driver</value>
  </property>
  <property name="url">
    <value>jdbc:mysql://localhost:3306/springweb_auth_db</value>
  </property>
  <property name="username">
    <value>j2ee</value>
  </property>
  <property name="password">
    <value>password</value>
  </property>
</bean>
```

Now all you need is the database specification: the tables. There are two ways to go here. The hard way involves using custom tables and thus specifying custom queries and rowmappers for the `AuthenticationDao`. This is beyond the scope of this tutorial. The documentation included in the full Acegi framework download contains more information on using a user database.

The easy way is to create a database tailored to Acegi's wishes, which would mean the following schema:



The “ENABLED” column could also be a `CHAR` or `VARCHAR` holding the values “true” and “false”. Obviously, the `AUTHORITY` column would hold the role names. Populate the database and you are all set!

6 To conclude

This tutorial was intended to simplify your understanding of how to use the Spring Acegi security framework to secure web applications when a standard J2EE security facility is either unavailable or undesirable. As you may have noticed, implementing Acegi security is not trivial. It *does* provide a fully portable and configurable security implementation, which renders your application completely independent of any built-in security mechanism your application server may provide. This enables you to switch application servers (e.g. for reasons of cost or scalability) without any modification to the security implementation. This may or may not be an issue for you, depending on how likely an application server change will be.

Another advocated advantage of Acegi security (with thanks to my esteemed colleague Levente Bokor for the input), is that it leverages one huge benefit of the Spring IoC container implementation: the ability to change the injected dependencies at runtime, programmatically. This would enable you to change the authentication and authorization parameters in case of a change in the security model used. Examples could be a change in role structure, which would translate into adjustments in the authorized role settings of authorized resources. Those adjustments could then be made without any redeployment of configuration files. However, I would tend to think that such high-impact adjustments would result from structural changes in the application which would necessitate redeployment anyway, which would enable you to make changes to the XML configuration along with it. So these runtime capabilities do not make any real difference from a security-configuration perspective (they may be advantageous from other perspectives, though).

I would definitely recommend using this framework for web applications with light to moderate security requirements for deployment on simple application servers. You automatically get complete portability to another application server environment. For heavy customization and advanced features, and if you are not likely to change your application server environment, I would recommend deploying on a more advanced application server with its own implementation of the J2EE security mechanisms.

7 Resources

Acegi Security homepage: <http://www.acegisecurity.org>.

Spring Framework homepage: <http://www.springframework.org>.

Eclipse Web Tools Project IDE: <http://www.eclipse.org/webtools>.

Tomcat Application Server: <http://tomcat.apache.org>.

MySQL database: <http://www.mysql.com>.

The book “Spring in Action” by Crag Walls & Ryan Breidenbach, Manning Publications.

The tutorial was written on OpenOffice Writer and OpenOffice Draw. Download it from <http://www.openoffice.org>.