

Sécurisation d'une application Web avec Acegi Security

par Baptiste Meurant (baptiste-meurant.developpez.com)

Date de publication :

Dernière mise à jour : 25/10/2007

Ce tutoriel a pour objectif de présenter les principaux aspects du framework de sécurité Acegi Security et son intégration à une application Web existante reposant sur DWR, Spring et Hibernate.

- I - Introduction
- II - Les différents aspects de la sécurité Web
 - A - Authentification des utilisateurs
 - B - Autorisations
 - 1 - Sécurisation des accès à une page web
 - 2 - Sécurisation des appels à une méthode
 - 3 - Sécurisation de la manipulation des objets du modèle
- III - Mise en place du projet
- IV - Installation et configuration d'Acegi
- V - Authentification
- VI - Autorisations
- VII - Sécurisation des urls
- VIII - Sécurisation des appels de méthodes
- IX - Conclusion
- X - Liens
- XI - Remerciements
- XII - Dans la même série ...

I - Introduction

Ce tutoriel, comme le précédent concernant DWR, n'a pas pour objectif de présenter Acegi Security de manière exhaustive mais ses principaux aspects et son intégration à une application Web existante reposant sur DWR, Spring et Hibernate. Il fait suite au [tutoriel DWR, Tapestry5, Spring et Hibernate](#).

Acegi Security est un **sous projet de la communauté Spring** très largement répandu et utilisé dans le monde de l'open source Web. Il propose des mécanismes de sécurité de manière intégrée aux environnements basés sur Spring, allant des plus simples aux plus complexes.

Avant de véritablement commencer ce tutoriel, nous allons effectuer quelques rappels sur les notions clefs de la sécurité d'une application Web et présenter brièvement la manière dont Acegi propose d'y répondre ainsi que les avantages de ce framework. Nous nous limiterons également à certains aspects d'Acegi parmi les plus utilisés sans explorer de fond en comble les possibilités de cet outil.

Les sources de ce tutoriel sont téléchargeables [ici](#) [miroir [http](#)]

II - Les différents aspects de la sécurité Web

Afin de mieux comprendre les différents aspects que nous allons traiter par la suite, voici une brève description des principales problématiques de sécurité communément traitées dans le monde du Web et de la manière dont Acegi permet d'y répondre.

Ces aspects se répartissent en deux catégories : l'**authentification** qui consiste à garantir que la personne connectée est bien celle qu'elle prétend être et les **autorisations** qui consistent à vérifier que la personne connectée a bien les permissions d'effectuer une action donnée.

A - Authentification des utilisateurs

La première étape de la sécurisation d'une application Web est l'authentification des utilisateurs qui la manipulent. En effet, contrairement à un site web, généralement ouvert à tous, les applications Web nécessitent généralement que l'identité de l'utilisateur soit connue, ne serait-ce que pour permettre la personnalisation des services qui lui sont offerts. Pour cela, il est nécessaire de mettre en place un mécanisme de login/password permettant ensuite de manipuler les informations propres à l'utilisateur, ses droits, éventuellement ses groupes, etc. Cette authentification se fait à partir d'informations stockées dans une base de données, un annuaire LDAP, etc.

Acegi propose une **solution native complète et totalement intégrée** des aspects authentification.

B - Autorisations

1 - Sécurisation des accès à une page web

Il s'agit de la première couche de sécurité, la plus simple à mettre en œuvre et la plus évidente : limiter les accès à une url aux utilisateurs d'une certaine catégorie. Par exemple, il paraît évident, dans le cadre d'une application nécessitant une authentification, que l'on doit interdire l'accès des pages de l'application aux utilisateurs non connectés - sans quoi l'authentification est parfaitement inutile. Il peut également être nécessaire de restreindre l'accès à certaines urls à un groupe très particulier d'utilisateurs connectés disposant de certains droits. C'est le cas de la partie administration d'une application web par exemple.

La mise en œuvre de cette sécurisation peut se faire à différents niveaux. On peut par exemple sécuriser les accès dans apache mais Acegi propose des mécanismes autrement plus complets, aboutis et portables que les mécanismes standard d'Apache. Acegi security permet une sécurisation fine des urls par un mécanisme de filtres de servlets évolués. A noter également que cette sécurisation est intégrée à la norme J2EE mais qu'Acegi en fournit une implémentation plus complète comme on le verra plus loin.

Remarque : l'utilisation de filtres de servlets impose de se baser sur les urls et contraint donc le nom et l'arborescence des pages pour gérer ces aspects. Attention donc aux règles de nommage.

2 - Sécurisation des appels à une méthode

C'est le deuxième aspect de la gestion des autorisations. Il s'agit de renforcer la sécurité en effectuant un contrôle au niveau de la couche de services : seuls les utilisateurs dûment authentifiés et disposant des droits nécessaires auront la possibilité d'exécuter une méthode sécurisée de cette façon. Si toute la couche de services est sécurisée de cette manière, cela peut permettre également de publier sans risques cette couche à un niveau plus visible - de manière, par exemple, à la mettre à disposition d'une autre application.

Acegi propose cette sécurisation par un ensemble de mécanismes élégants et non intrusifs basés sur des Dynamic Proxy java et les concepts de POA (Programmation Orientée Aspect) grâce à Spring AOP. On peut noter que l'intégration d'un autre framework de POA - AspectJ est également possible (d'autant qu'interface21 - la société qui se cache derrière Spring - a récemment récupéré AspectJ dans son giron). Cependant l'intégration d'AspectJ est plus complexe (mais plus performante) et nécessite d'avantages de connaissances AOP. Dans le cadre de ce tutoriel, nous n'aborderons pas AspectJ.

Attention cependant au fait que la mise en oeuvre d'un tel niveau de sécurité est, bien entendu, loin d'être anodin en terme de complexité et donc, dans une moindre mesure, de performances. Il s'agit alors de bien réfléchir à la pertinence d'un tel niveau de sécurité. En outre, tout comme la sécurisation des urls impose dès le départ de prendre en compte cet aspect, la sécurisation des appels de méthode peut entraîner, dans certains cas, la modification de l'architecture de l'application (création d'une couche sécurité dédiée pour éviter les injections de beans cycliques).

Nous verrons par la suite une implémentation simple d'une sécurisation de ce type.

3 - Sécurisation de la manipulation des objets du modèle

C'est le dernier point de sécurité que nous aborderons ici. C'est l'un des plus complexes à mettre en #uvre et de loin le plus coûteux en terme de performances. Tout comme nous avons sécurisé les appels de méthodes, il s'agit ici de sécuriser les instances d'objets du modèle. Dans un contexte d'application e-commerce, par exemple, un utilisateur n'a le droit de manipuler que les objets du modèle le concernant (son panier par exemple).

Mon expérience personnelle montre que ce niveau de sécurité est rarement utilisé car il correspond à la nécessité de répondre à des exigences de sécurité très fortes et nécessite un investissement plus important. Cet aspect ne sera pas traité dans le cadre de ce tutoriel (Tout du moins dans sa première version).

De manière générale, gérer ces différents aspects par Acegi apporte une portabilité complète, une indépendance vis-à-vis du serveur d'application utilisé. Acegi étend également de manière significative les mécanismes J2EE standard d'authentification et de sécurisation des urls tels que l'authentification automatique limitée dans le temps, la détection du double login, etc. Enfin, Acegi propose - par ses mécanismes de POA et l'utilisation de Spring AOP une gestion unique de la sécurisation des couches service et model.

La suite de cet article propose une implémentation simple des trois premiers point évoqués ci-dessus : l'authentification, la sécurisation des urls et la sécurisation des appels de méthodes.

III - Mise en place du projet

Si vous avez suivi le **tutoriel précédent** traitant de DWR, Tapestry, Spring et Hibernate :

- Copier ce projet depuis votre workspace Eclipse et le copier dans ce même workspace en changeant le nom.

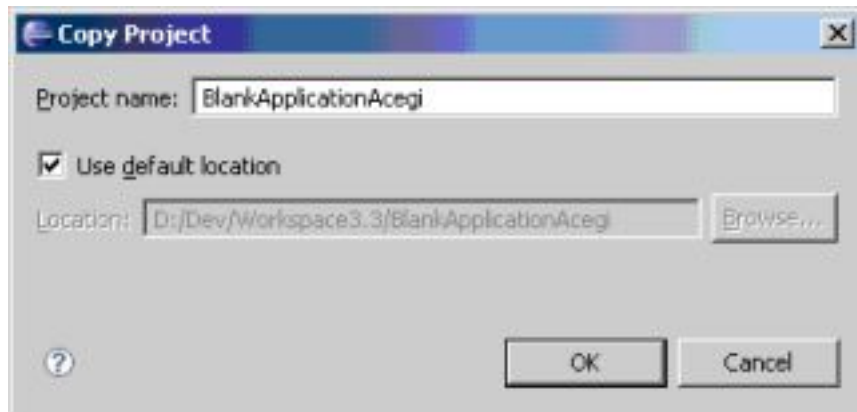


Figure 1 : Copie du projet

- Allez ensuite dans les propriétés du projet, Web Project Settings et changer le nom du contexte.

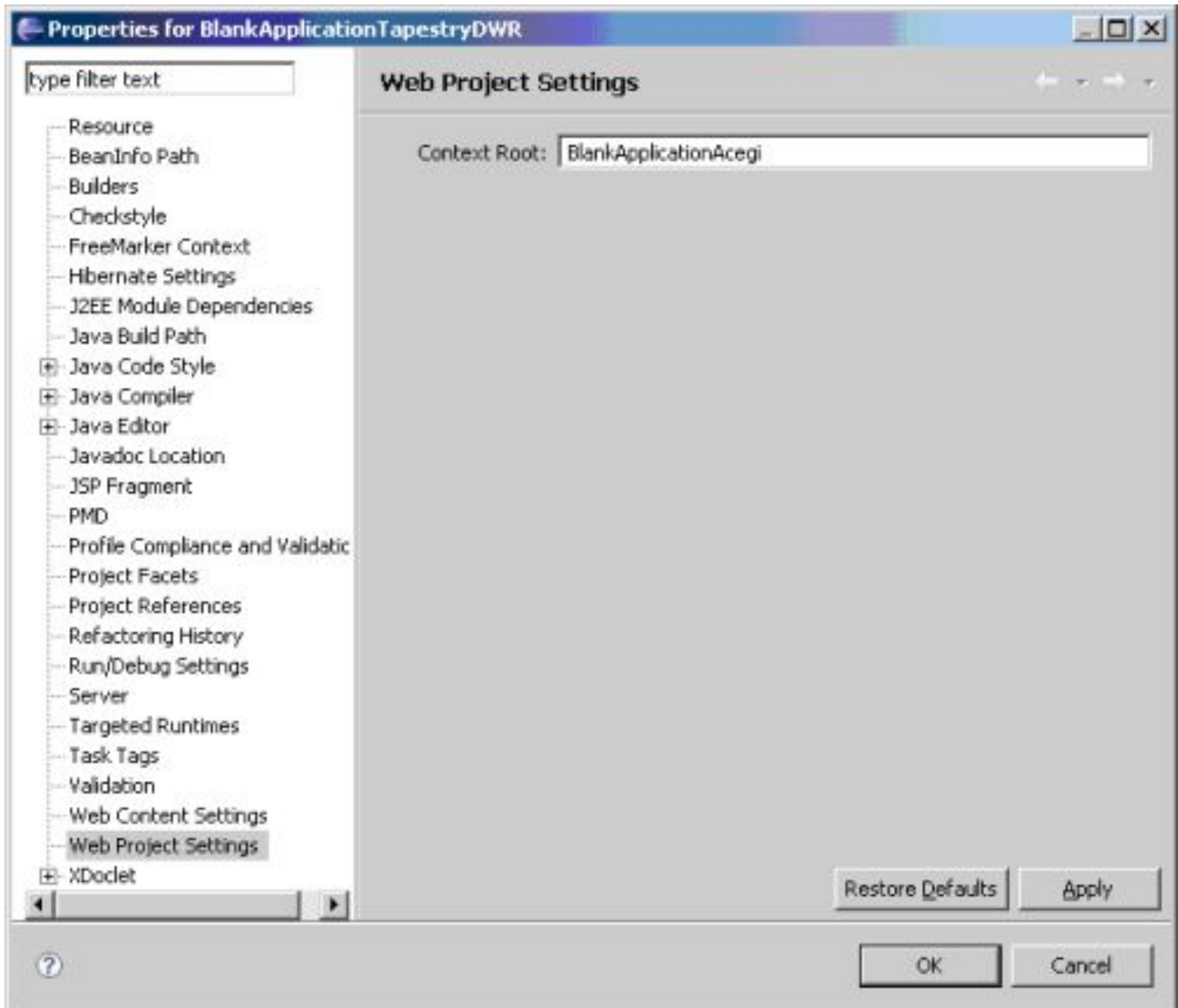


Figure2 : Changement du contexte

Editer ensuite le fichier `org.eclipse.wst.common.component` dans le répertoire `.settings` du nouveau projet. Modifier l'élément `wb-module deploy-name` en changeant le nom du projet dupliqué par le nom du nouveau projet. Cela évite toute confusion dans la vue `Server` d'Eclipse (sans cela, c'est toujours l'ancien nom qui s'affiche mais cela n'affecte pas le fonctionnement). Relancer ensuite Eclipse.

Si vous n'avez pas suivi le tutoriel précédent :

- Récupérer les sources du tutoriel DWR, Tapestry5, Spring Hibernate [ici](#). Le dézipper sur votre machine.
- Créer un nouveau projet en suivant la procédure décrite dans les chapitres II et III du deuxième tutoriel DWR, Tapestry5, Spring, Hibernate disponible [ici](#).
- Importer l'ensemble du tutoriel (sources, lib, etc) précédent dans ce nouveau projet.
- Redéfinir le répertoire config comme source folder du projet.

A l'issue de ces étapes, le projet doit ressembler à cela :

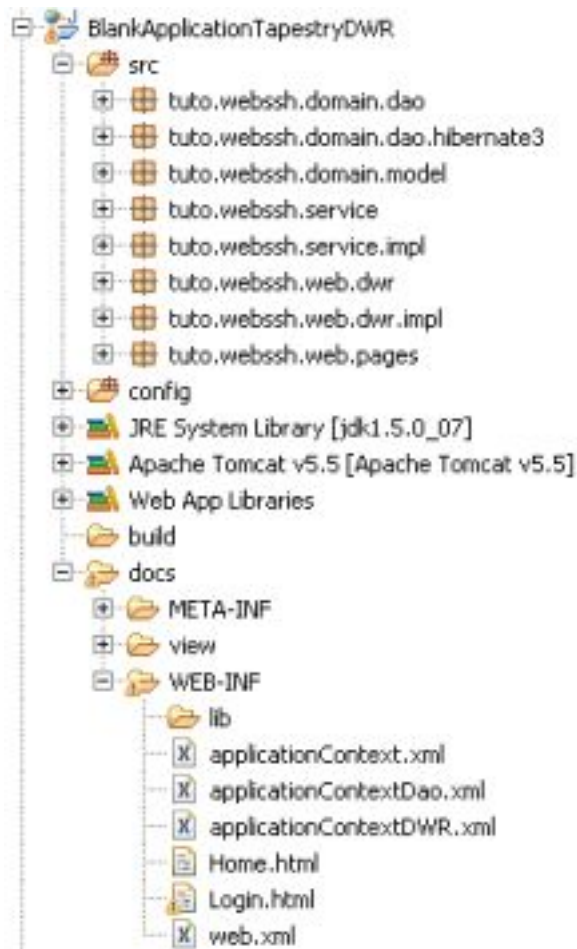


Figure3 : Projet vide

Dans les deux cas :

- Associer ce projet avec le serveur Tomcat.
- Dans le fichier Home.html, modifier les liens d'inclusion des fichiers Javascript dans les pages html pour adapter le contexte de l'application au contexte du nouveau projet :

```

<script type='text/Javascript' src='/<contexteProjet>/dwr/interface/UserDWR.js'></script>
<script type='text/Javascript' src='/<contexteProjet>/dwr/engine.js'></script>
<script type='text/Javascript' src='/<contexteProjet>/dwr/util.js'></script>
    
```

IV - Installation et configuration d'Acegi

Avant quelque implémentation que ce soit, il est nécessaire d'effectuer l'installation et la configuration globale d'Acegi security.

- Télécharger la distribution complète d'Acegi [ici](#).
- La décompresser et copier le jar acegi-security-1.x.x.jar dans WEB-INF/lib.

On doit maintenant ajouter la configuration d'Acegi dans le web.xml :

- Ajouter le filtre Acegi
- Editer le fichier web.xml, ajouter le mapping du filtre Acegi. Attention à l'ordre du mapping, le filtre sessionInView doit être le premier filtre exécuté et le filtre Acegi doit se trouver avant le filtre Tapestry. L'ordre est donc le suivant : filtre sessionInView puis filtre Acegi puis filtre Tapestry :

Intégration d'Acegi dans le web.xml

```
<filter>
  <filter-name>Acegi Security Filter</filter-name>
  <filter-class>org.acegisecurity.util.FilterToBeanProxy</filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>org.acegisecurity.util.FilterChainProxy</param-value>
  </init-param>
</filter>

<filter>
  <filter-name>Hibernate Session In View Filter</filter-name>
  <filter-class>
    org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
  </filter-class>
</filter>

<filter>
  <filter-name>app</filter-name>
  <!-- Special filter that adds in a T5 IoC module derived from the Spring WebApplicationContext.
  -->
  <filter-class>
    org.apache.tapestry.spring.TapestrySpringFilter
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>Hibernate Session In View Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>Acegi Security Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

<filter-mapping>
  <filter-name>app</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

En effet, cela permet de bénéficier de lazy loading Hibernate dès la phase d'authentification, avant, donc l'exécution du filtre acegi. Nous aurons d'ailleurs besoin de cette fonctionnalité comme on le verra par la suite pour récupérer les droits de l'utilisateur, configurés en lazy (voir le **premier tutoriel** [tapestry5](#), Spring, Hibernate pour des détails sur cette notion).

Dans ce cas, le filtre est configuré de manière à ce que toutes les urls soient soumises à la validation d'Acegi : toute url entrée déclenchera le filtre. C'est ensuite Acegi qui appliquera un second niveau de filtres chaînés pour affiner les différents niveaux de sécurité. Ainsi, le filtre FilterChainProxy défini dans le web.xml permettra de lier les différents filtres de sécurité grâce à une configuration Spring classique.

V - Authentification

Quels que soient les aspects de la sécurité que l'on souhaite traiter dans Acegi, il est indispensable d'effectuer préalablement la configuration du FilterChainProxy référencé dans le web.xml plus haut. Nous allons commencer par configurer ce proxy afin de déclarer l'authentification des utilisateurs. En effet, la configuration du web.xml précédente renvoie l'ensemble des requêtes vers le Proxy de filtres Acegi. C'est donc ce dernier que nous devons configurer avec les différents types de filtres existants pour mettre en place la sécurisation des pages telle que nous l'attendons.

- Créer un nouveau fichier applicationContextSecurity.xml de configuration Spring dans WEB-INF. le fait de créer un nouveau fichier n'est pas obligatoire mais, en fonction du niveau de sécurité attendu, il peut atteindre une taille conséquente et il est fortement conseillé de le traiter de manière autonome :

Création du fichier applicationContextSecurity.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!-- ===== FILTER CHAIN ===== -->
<bean id="filterChainProxy"
class="org.acegisecurity.util.FilterChainProxy">
<property name="filterInvocationDefinitionSource">
<value>
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
PATTERN_TYPE_APACHE_ANT
/login=
                httpSessionContextIntegrationFilter
                /login.form=
                httpSessionContextIntegrationFilter
            /assets/**=
                httpSessionContextIntegrationFilter
            /j_acegi_security_check=
                httpSessionContextIntegrationFilter,
                formAuthenticationProcessingFilter
            /**=
                httpSessionContextIntegrationFilter,
                exceptionTranslationFilter

</value>
</property>
</bean>
</beans>
```

Grâce à cette configuration, on définit successivement :

- Que les url relatives `/login` et `/login.form` ne nécessitent pas d'être authentifié. En effet, le filtre `httpSessionContextIntegrationFilter` est le seul filtre obligatoire, essentiel au fonctionnement d'acegi. Aucun autre filtre n'est défini, ce qui signifie qu'aucune authentification préalable n'est nécessaire. Cela paraît plutôt logique puisque c'est justement ces pages qui permettent de se logguer (l'url `login.form` est un mécanisme interne à Tapestry).
- Il en est de même pour les urls commençant par `/assets/`. Il s'agit des traitements javascripts et des images appelées de manière transparente par acegi pour gérer la validation des formulaires (champs obligatoires, etc.) Sans cette configuration qui spécifie que l'inclusion de ces fichiers javascript ne nécessite pas d'être authentifié, la page de login ne s'affiche pas correctement.
- Que l'url relative `/j_acegi_security_check` correspond à l'action appelée lors d'une tentative d'authentification. En effet, en plus du filtre obligatoire `httpSessionContextIntegrationFilter`, cette url est associée au filtre `formAuthenticationProcessingFilter` ce qui signifie qu'il s'agit d'une url permettant d'authentifier un utilisateur.

Concrètement, c'est cette url qui est appelée lors de la validation du formulaire de login (page *login*) ce qui a pour action de déclencher la tentative d'authentification comme on le verra plus loin.

Ces filtres sont appliqués de manière séquentielle ; c'est-à-dire que lorsqu'une url est entrée, acegi regardera successivement si il s'agit de l'url login puis, sinon, si il s'agit de l'url `j_acegi_security_check` et enfin appliquera le traitement par défaut dans les autres cas (en l'occurrence rien mais cela sera modifié par la suite). Attention donc à l'ordre de définition car si les url sont inversées il sera rigoureusement impossible de s'authentifier sur votre application !

Les deux filtres cités dans le paragraphe précédent ainsi que le filtre `filterSecurityInterceptor` que l'on configurera plus loin sont les seuls filtres rigoureusement nécessaires au fonctionnement d'acegi. Il existe néanmoins d'autres filtres remplissant chacun une fonction particulière. La liste ci-dessous récapitule très brièvement chacun des filtres existant et leur fonction. La liste est présentée dans l'ordre d'exécution des filtres. Pour plus d'information, se référer à la documentation d'acegi :

- **ChannelProcessingFilter** : change le protocole : redirige l'url de http vers https.
- **ConcurrentSessionFilter** : bloque l'appel dans le cas où une session est déjà ouverte pour cet utilisateur.
- **HttpSessionContextIntegrationFilter** : stocke le contexte acegi dans la session. Ce filtre est obligatoire.
- Filtres d'authentification - **AuthenticationProcessingFilter**, **CasProcessingFilter**, **BasicProcessingFilter**, **HttpRequestIntegrationFilter**, etc. : permettent d'exécuter les actions d'authentification des utilisateurs.
- **SecurityContextHolderAwareRequestFilter** : autorise l'utilisation de l'API de sécurité J2EE standard.
- **RememberMeProcessingFilter** : permet une authentification automatique d'un utilisateur pendant une période de temps prédéfinie.
- **AnonymousProcessingFilter** : si ce filtre est défini, autorise une connexion anonyme si l'utilisateur n'a pas encore été authentifié par les filtres précédents.
- **ExceptionHandler** : catch les exceptions acegi afin de retourner une erreur http standard ou d'exécuter un traitement particulier.
- **FilterSecurityInterceptor** : protège les urls.

Chacun des filtres utilisés dans notre configuration doit donc maintenant être défini et paramétré dans notre fichier `applicationContextSecurity.xml`.

- La définition du filtre **HttpSessionContextIntegrationFilter** se contente d'associer la classe d'implémentation du filtre.

Définition du filtre `HttpSessionContextIntegrationFilter`

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

- La définition du filtre **ExceptionHandler** associe la classe d'implémentation du filtre et injecte une dépendance au bean définissant le point d'entrée (et d'authentification) de l'application de manière à pouvoir rediriger vers ce point les éventuelles exceptions acegi : toute erreur ramènera l'utilisateur sur la page de connexion. Le bean **formLoginAuthenticationEntryPoint** définit l'url considérée comme le point d'entrée de l'application ainsi qu'un paramètre permettant de forcer le protocole à https (en phase de développement, ce paramètre peut rester à false mais doit être passé à true en production pour que les paramètres d'authentification soient cryptés).

Définition du filtre ExceptionTranslationFilter

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

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

- Le filtre **formAuthenticationProcessingFilter** définit l'ensemble de la configuration de l'authentification. Outre la classe d'implémentation du filtre, on peut remarquer que l'on spécifie l'url de redirection en cas d'échec ou de succès (respectivement login et home) ainsi que l'url permettant de déclencher l'action d'authentification à proprement parler. Enfin, la dernière propriété du bean est une référence (une injection donc) du bean permettant la gestion de l'authentification elle-même :
 - 1 La propriété **userDetailsService** faisant référence au bean de même nom injecte dans le provider le dernier objet permettant la configuration de l'authentification : l'implémentation qui contiendra la méthode d'authentification. Cela peut être une implémentation standard acegi (par exemple *org.acegisecurity.userdetails.jdbc.JdbcDaoImpl*) ou une implémentation personnalisée mais dans ce cas la classe doit implémenter l'interface *org.acegisecurity.userdetails.UserDetailsService*. C'est cette dernière méthode que nous allons implémenter afin de mieux comprendre le mécanisme. Dans notre cas il s'agit de *tuto.webssh.security.UserDetailsServiceImpl* qui, au passage, a besoin du bean *userManager* déjà défini par ailleurs.
 - 2 Le bean **authenticationManager** définit la liste des providers, c'est-à-dire la liste des beans qui seront chargés de réaliser effectivement l'opération d'authentification. Il peut en effet exister plusieurs providers différents - la stratégie d'interconnexion de ces providers est alors à définir (consulter la documentation acegi). Dans notre cas, nous n'avons qu'un seul fournisseur d'authentification, le bean *authenticationProvider*.
 - 3 Le bean **authenticationProvider** définit d'une part sa propre classe d'implémentation, *DaoAuthenticationProvider*, ce qui signifie qu'il s'agit d'une authentification basée sur l'appel d'une méthode d'un Dao. A noter que les stratégies d'authentification les plus couramment utilisées sont :
 - a Une implémentation via **base de données** : les paramètres de connexion ainsi que les requêtes de récupération des informations des utilisateurs sont alors configurables en xml dans ce même fichier (voir la documentation acegi). Acegi fournit par défaut un schéma de base de données mais il est entièrement configurable. Au cas où l'on utiliserait le schéma proposé par acegi, la classe *JdbcDaoImpl* permet de le manipuler sans avoir besoin de créer son propre objet *UserDetailsService*. Dans la suite de cet article, nous souhaitons cependant prendre en compte le fait qu'il est fréquent qu'une application (existante ou non) impose son propre modèle de données (voir un modèle mixte LDAP/BDD) en matière de gestion de la sécurité. Nous traiterons donc le cas le plus souple en implémentant notre propre objet *UserDetailsService*.
 - b Une implémentation via **LDAP** permettant de récupérer un objet user ainsi que ses groupes et ses droits. Les paramètres d'accès ainsi que les paramètres de mapping entre l'objet LDAP et l'objet Java sont configurables en xml dans ce même fichier (voir documentation acegi).

Définition du filtre formAuthenticationProcessingFilter

```
<!-- ===== formAuthenticationProcessingFilter ===== -->
```

Définition du filtre formAuthenticationProcessingFilter

```

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

<!-- ===== authenticationManager ===== -->
<bean id="authenticationManager"
  class="org.acegisecurity.providers.ProviderManager">
  <property name="providers">
    <list>
      <ref local="authenticationProvider"/>
    </list>
  </property>
</bean>

<!-- ===== authenticationProvider ===== -->
<bean id="authenticationProvider"
  class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userService">
    <ref bean="userService"/>
  </property>
</bean>

<!-- ===== userService ===== -->
<bean id="userService"
  class="tuto.webssh.security.UserDetailsServiceImpl">
  <property name="userManager">
    <ref bean="userManager"/>
  </property>
</bean>

```

- Nous allons donc créer la classe d'authentification **UserDetailsServiceImpl** dans un nouveau package *tuto.webssh.security* telle que nous l'avons définie dans notre fichier xml. Cette classe doit implémenter l'interface acegi *UserDetailsService*.

Classe UserDetailsServiceImpl

```

package tuto.webssh.security;

import java.util.Set;

import org.acegisecurity.GrantedAuthority;
import org.acegisecurity.GrantedAuthorityImpl;
import org.acegisecurity.userdetails.UserDetails;
import org.acegisecurity.userdetails.UserDetailsService;
import org.acegisecurity.userdetails.UsernameNotFoundException;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.dao.DataAccessException;

import tuto.webssh.domain.model.Rights;
import tuto.webssh.domain.model.User;

```

Classe UserDetailsServiceImpl

```
import tuto.webssh.service.UserManager;

/**
 * Implements a strategy to perform authentication.
 * @author bmeurant
 */
public class UserDetailsServiceImpl implements UserDetailsService {

    private UserManager userManager;
    private final Log logger = LoggerFactory.getLog(UserDetailsServiceImpl.class);

    /**
     * setter to allows spring to inject userManager implementation
     * @param userManager : object (implementation of UserManager interface) to inject.
     */
    public void setUserManager(UserManager userManager) {
        this.userManager = userManager;
    }

    /**
     * Get the user object corresponding to the given login, check the password stored
     * into the secured session context and grant corresponding rights for the user.
     * @param login: user login
     * @return an UserDetails object representing the authenticated user and his rights
     * if the authentication is successful.
     */
    public UserDetails loadUserByUsername(String login) {
        logger.info("Trying to Load the User with login: "+login+" and password [PROTECTED] from database
and LDAP directory");
        try{
            logger.info("Searching the user with login: "+login+" in database");
            User user = userManager.getUser(login);

            if (null == user) {
                logger.error("User with login: "+login+" not found in database. Authentication failed for user
"+login);
                throw new UsernameNotFoundException("user not found in database");
            }
            logger.info("user with login: "+login+" found in database");

            Set<Rights> rights = user.getRights();
            GrantedAuthority[] arrayAuths = new GrantedAuthority[rights.size()+1];
            int i=0;
            arrayAuths[i++] = new GrantedAuthorityImpl("ROLE_AUTH");

            for (Rights right : rights) {
                arrayAuths[i++] = new GrantedAuthorityImpl("ROLE_"+right.getLabel());
            }

            logger.debug("Create User for acegi features for User with login: "+login);
            org.acegisecurity.userdetails.User acegiUser =
                new org.acegisecurity.userdetails.User(login,user.getPasswordUser(),true, true, true, true,
arrayAuths);
            logger.info("user with login: "+login+" authenticated");

            return acegiUser;
        }
        catch (DataAccessException e){
            logger.error("Cannot retrieve Data from Database server : "+e.getMessage()+". Authentication
failed for user "+login);
            throw new UsernameNotFoundException("user not found", e);
        }
    }
}
```

On récupère donc ici le user à partir de son login puis on renseigne les objets acegi User et GrantedAuthority qui stockent respectivement les informations de l'utilisateur et ses droits. La correspondance entre le mot de passe saisi et celui stocké sera vérifiée par acegi de manière automatique et transparente et une exception de type *UsernameNotFoundException* levée en cas d'incohérence, ce qui aura pour effet d'invalider l'authentification. On remarque que cette exception est également levée si l'utilisateur n'existe pas ou si une erreur d'accès aux données est survenue.

En ce qui concerne les droits, on récupère l'ensemble des droits de l'utilisateur stockés en base que l'on préfixe par ROLE_ par convention. On ajoute aussi systématiquement le rôle ROLE_AUTH dès que l'utilisateur existe de manière à pouvoir gérer de manière plus globale les droits d'accès les plus simples.

- Nous allons ensuite modifier la classe Login.java afin que la validation du formulaire déclenche l'authentification acegi.

Modification de la classe Tapestry Login

```
package tuto.webssh.web.pages;

import org.apache.tapestry.Link;
import org.apache.tapestry.annotations.ApplicationState;
import org.apache.tapestry.annotations.Inject;
import org.apache.tapestry.annotations.Persist;
import org.apache.tapestry.beaneditor.Validate;
import org.apache.tapestry.internal.services.LinkImpl;
import org.apache.tapestry.services.Request;
import org.apache.tapestry.services.Response;

public class Login {

    private static final String BAD_CREDENTIALS =
        "Bad login and/or password. Please retry.";

    @Persist
    private boolean error = false;

    @ApplicationState
    private String login;

    @Inject
    private Request request;

    @Inject
    private Response response;

    private String password;

    public String getLogin() {
        return login;
    }

    @Validate("required")
    public void setLogin(String login) {
        this.login = login;
    }

    public String getPassword() {
        return password;
    }

    public String getErrorMessage() {
        String ret = null;
        if (error) {
            ret = BAD_CREDENTIALS;
        }
    }
}
```

Modification de la classe Tapestry Login

```
}
return ret;
}

@Validate("required")
public void setPassword(String password) {
    this.password = password;
}

Link onSuccess() {
    Link link= new LinkImpl(response, request.getContextPath(), "j_acegi_security_check");
    link.addParameter("j_username", login);
    link.addParameter("j_password", password);
    return link;
}
}
```

On a donc modifié la méthode *onSuccess* pour qu'elle retourne un lien Tapestry au lieu d'une String et pour que le post du formulaire pointe sur l'url d'authentification `acegi_j_acegi_security_check`. On note que la définition du lien nécessite la création des deux attributs `request` et `response`, récupérés depuis le contexte J2EE par l'annotation `@Inject`. On a également supprimé au passage l'attribut `requestGlobals` devenu inutile ainsi que l'injection du bean `userManager` qui nous permettait avant de récupérer l'utilisateur depuis la BDD - nous le récupérons désormais depuis la session.

- Inclure le nouveau fichier de configuration au `web.xml` :

Référencement du fichier de configuration acegi dans le web.xml

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext.xml /WEB-INF/applicationContextDao.xml
  /WEB-INF/applicationContextDWR.xml WEB-INF/applicationContextSecurity.xml
</param-value>
</context-param>
```

En lançant le serveur et en testant le login avec les bons paramètres puis avec un mauvais password on s'aperçoit de plusieurs choses :

- Une authentification réussie redirige bien l'utilisateur vers la page *home*, comme définit dans le *formAuthenticationProcessingFilter*.
- Un échec redirige bien l'utilisateur vers la même page de *login*, comme définit dans le *formAuthenticationProcessingFilter*.
- La vérification du mot de passe est effectuée sans que nous l'ayons explicitement implémentée. C'est bien acegi qui s'en est chargé, de manière totalement transparente. Cela a également pour avantage de ne pas nécessiter de manipulations du password lui-même et de conserver cet objet password dans le contexte de session sécurisé d'acegi.

Une dernière chose reste à configurer concernant l'authentification. En effet, en cas d'échec de la connexion, l'utilisateur est redirigé vers la page login d'où il vient sans message d'erreur explicite, ce qui peut être perturbant. Comme nous avons déjà implémenté ce mécanisme dans le tutoriel précédent, le code est déjà prêt et il ne reste plus qu'à passer le paramètre `error` à `true` dans l'url de redirection.

- Pour cela on doit modifier le bean *formAuthenticationProcessingFilter* dans le fichier *applicationContextSecurity.xml* :

Modification du formAuthenticationProcessingFilter

```
<!-- ===== formAuthenticationProcessingFilter ===== -->
<bean id="formAuthenticationProcessingFilter"
class="org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">
  <property name="authenticationManager">
    <ref bean="authenticationManager" />
  </property>
  <property name="authenticationFailureUrl">
    <value>/login?error=true</value>
  </property>
  <property name="defaultTargetUrl">
    <value>/home</value>
  </property>
  <property name="filterProcessesUrl">
    <value>/j_acegi_security_check</value>
  </property>
</bean>
```

On doit également modifier la page Login.java afin qu'elle puisse récupérer le paramètre. Dans ce cas précis on doit utiliser l'objet request J2EE et non le mécanisme interne Tapestry car il ne s'agit pas d'un paramètre Tapestry. Cependant cette solution ne doit en aucun cas être utilisée pour passer des paramètres entre deux composants Tapestry. On crée donc une méthode onActivate qui, grâce à son nom, sera appelée à chaque chargement de la page. On renseigne ensuite la propriété error à partir de la valeur du paramètre passé (false si null), ce qui permet d'afficher le message d'erreur sur la page *Login.html* grâce à la méthode *getErrorMessage()*. A noter qu'au lieu d'appeler la méthode *onActivate* on aurait pu la nommer d'une autre façon et placer devant l'annotation *@onActivate*.

Modification de Login.java

```
void onActivate() {
  error = new Boolean(request.getParameter("error")).booleanValue();
}
```

Après relance du serveur on peut constater qu'un échec de connexion redirige désormais toujours vers la page de login mais que le message d'erreur est affiché.

VI - Autorisations

Nous venons de mettre en place une solution d'authentification sur notre application Web. Nous allons maintenant nous intéresser au deuxième aspect de la gestion de la sécurité : la gestion des autorisations.

Dans acegi, la gestion des autorisations est un mécanisme très évolué et, du fait, assez complexe dans ses principes comme dans sa mise en œuvre. Cependant, un certain nombre de configurations par défaut permettent, dans la plupart des cas simples, de mettre en place les autorisations à moindre frais. Dans ce chapitre, nous allons brièvement passer en revue les mécanismes généraux de gestion des autorisations dans acegi. Les principes évoqués ci-dessous seront valables pour la protection de n'importe quelle ressource, qu'il s'agisse d'une url, d'une méthode, d'un objet.

Afin de déterminer si un utilisateur peut accéder ou non à une ressource, acegi utilise un système de vote. Pour ce faire, le framework met à disposition une hiérarchie de classes dans le package `org.acegisecurity.vote` comprenant des classes de voter abstraites et quelques implémentations standard. Un voter peut effectuer quatre actions en fonction des attributs de configuration définis :

- **Ne rien faire** si les attributs de configuration ne correspondent pas à ceux pour lesquels il vote (le voter n'est pas appelé).
- **S'abstenir** (return `ACCESS_ABSTAIN`) si le voter n'a pas d'opinion.
- **Voter pour** (return `ACCESS_GRANTED`) si il estime que l'utilisateur a le droit d'accéder à la ressource protégée.
- **Voter contre** (return `ACCESS_DENIED`) si il estime que l'utilisateur n'a pas le droit d'accéder à la ressource protégée.

La participation ou non d'un voter à un vote, c'est-à-dire au processus d'autorisation pour un utilisateur d'accéder à une ressource est déterminée par les attributs de configuration (Interface *ConfigAttribute*) associés à la ressource. Ainsi, un voter ne participe qu'aux votes concernant des ressources associées à certains attributs de configurations comme on le verra plus loin. Dans le cas contraire, le voter ne participe pas au vote.

Le ou les voters associés à un processus d'autorisation d'accès sont associés à travers un **AccessDecisionManager** qui, comme son nom l'indique, permettra de prendre une décision quant à l'accès à une ressource pour un utilisateur. Bien entendu, les voters peuvent être nombreux, chacun votant en fonction de ses propres critères. Il est donc nécessaire de déterminer quelle stratégie l'*AccessDecisionManager* doit appliquer :

- **AffirmativeBased** : L'accès est autorisé si au moins un voter l'a autorisé.
- **ConsensusBased** : L'accès est autorisé si la majorité des voters l'a autorisé.
- **UnanimousBased** : L'accès est autorisé si tous les voters l'on autorisé.

Il existe enfin des paramètres permettant éventuellement de déterminer la stratégie à appliquer en cas de *ConsensusBased* et d'égalité ainsi que la manière de traiter les abstentions (on verra un exemple plus loin).

Ce système est donc extrêmement sophistiqué et permet une très grande finesse dans la configuration des autorisations d'accès. Cependant, dans bien des cas (notamment concernant la sécurisation des url vue plus loin), la manipulation d'un voter unique est amplement suffisante.

VII - Sécurisation des urls

Nous allons maintenant fixer les différents concepts vus dans le chapitre précédent en mettant en place la sécurisation des URL de notre application. En effet, nous disposons d'une authentification sécurisée sur notre application grâce à acegi mais pour le moment cette authentification semble parfaitement inutile car ni la page home (la seule page de notre application exemple), ni les différentes interfaces DWR publiées ne sont sécurisées. Cela signifie que bien qu'une authentification soit mise en place, on peut accéder à toute l'application sans s'authentifier ce qui est, on en convient, un peu dommage # Nous allons donc remédier à cela en mettant en place la sécurisation des urls - car nous allons protéger non seulement l'ensemble des pages publiées de notre applications mais également tous les éléments inclus dans ces pages (javascripts notamment). En fait, dès qu'une url est demandée, directement ou indirectement, acegi vérifiera si la ressource est protégée ou non. Cela est extrêmement utile notamment pour les interfaces DWR qui publient des méthodes qui peuvent s'avérer sensibles dans un contexte anonyme.

- Modifier le **FilterChainProxy** afin d'ajouter le filtre *filterSecurityInterceptor* à toutes les urls dont le nom ne correspond pas aux précédents patterns (login, login.form, assets/**, etc.). Ainsi, l'ensemble ces urls seront protégées par ce filtre ce qui signifie qu'elles ne seront accessibles qu'aux utilisateurs authentifiés et disposant des autorisations nécessaires (cf. plus loin).

Modification du FilterChainProxy

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
<!-- ===== FILTER CHAIN ===== -->
<bean id="filterChainProxy"
class="org.acegisecurity.util.FilterChainProxy">
<property name="filterInvocationDefinitionSource">
<value>
CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
PATTERN_TYPE_APACHE_ANT
/login=
                httpSessionContextIntegrationFilter
            /login.form=
                httpSessionContextIntegrationFilter
        /assets/**=
                httpSessionContextIntegrationFilter
        /j_acegi_security_check=
                httpSessionContextIntegrationFilter,
                formAuthenticationProcessingFilter
        /**=
                httpSessionContextIntegrationFilter,
                exceptionTranslationFilter,
                filterSecurityInterceptor
    </value>
</property>
</bean>
</beans>
```

Le filtre *filterSecurityInterceptor* définit l'ensemble de la configuration de sécurisation des url. On peut remarquer qu'il fait référence à l'*authenticationManager* défini plus haut.

Définition du filterSecurityInterceptor

```
<!-- ===== filterSecurityInterceptor ===== -->
<bean id="filterSecurityInterceptor"
class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
<property name="authenticationManager">
```

Définition du filterSecurityInterceptor

```
<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
    /**=ROLE_AUTH
  </value>
</property>
</bean>
```

On remarque la propriété `objectDefinitionSource`. Cette propriété permet de configurer les différentes pages protégées de notre application. On note ici que les urls sont converties en minuscule afin d'éviter tout problème de casse et que la syntaxe des expressions régulière est celle de ANT, plus simple que la syntaxe standard java par défaut. Enfin, on note surtout que toutes les urls sont protégées par l'attribut de configuration `ROLE_AUTH`. Cela signifie que l'accès à ces ressources sera déterminé par la décision de l'`AccessDecisionManager` et que les voters dont la configuration correspond à 'ROLE_AUTH' participeront au vote.

- On définit ensuite le bean **`accessDecisionManager`** référencé par la propriété `accessDecisionManager` du `filterSecurityInterceptor`.

Définition du bean accessDecisionManager

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

Dans cette configuration, un seul voter est défini pour déterminer l'accès à la ressource, ce qui rend le choix du type d'`AccessDecisionManager` (ici `UnanimousBased`) peu déterminante.

- Enfin, nous allons définir et configurer notre unique voter.

Définition du voter

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

Dans notre cas, nous utilisons l'implémentation standard `RoleVoter`. Cette classe extrêmement simple participe à un vote si au moins un attribut de configuration (`ROLE_AUTH` dans notre cas) commence par le préfix configuré ('ROLE_' ici et par défaut). L'accès sera autorisé si l'utilisateur a au moins un objet de type `GrantedAuthority` exactement égal à l'un des attributs de configuration commençant par le préfix, sinon il votera contre l'accès. Dans le cas où aucun attribut de configuration ne commence par le préfix configuré, le voter s'abstiendra.

En bref, dans notre cas, le voter participe au vote car nous avons défini `ROLE_AUTH` comme attribut de configuration pour toutes nos url et que cet attribut commence par `ROLE_`, préfix défini dans la configuration du voter. Ce voter votera pour l'accès à toute page pour tout utilisateur authentifié car nous avons ajouté le `GrantedAuthority` `ROLE_AUTH` à toutes les identités connectées dans la classe de connexion `UserDetailsServiceImpl`.

Cette configuration de voter a le mérite d'être très simple à mettre en #uvre mais introduit malheureusement quelques confusions entre les attributs de configurations déterminant la participation d'un voter à un vote et le traitement contenu par le voter lui-même. **Attention donc à bien différencier ces deux notions** par la suite.

Testons maintenant cette nouvelle configuration :

- Démarrer le serveur et entrer l'url : <http://localhost:8080/BlankApplicationAcegi/home>. On s'aperçoit que l'on est automatiquement redirigé vers la page de login. En effet, comme toutes les pages de notre application, la page home est sécurisée. Lorsque -comme dans ce cas - un utilisateur non authentifié essaye d'y accéder, acegi refuse l'accès et redirige automatiquement vers la page définie comme le point d'entrée comme le montrent les logs d'acegi en DEBUG :

Logs de démarrage de l'application

```
45845 DEBUG org.acegisecurity.intercept.web.PathBasedFilterInvocationDefinitionMap
- Converted URL to lowercase, from: '/home'; to: '/home'
45845 DEBUG org.acegisecurity.intercept.web.PathBasedFilterInvocationDefinitionMap
- Candidate is: '/home'; pattern is /**; matched=true
45845 DEBUG org.acegisecurity.intercept.AbstractSecurityInterceptor
- Secure object: FilterInvocation: URL: /home; ConfigAttributes: [ROLE_AUTH]
45861 DEBUG org.acegisecurity.ui.ExceptionTranslationFilter
- Authentication exception occurred; redirecting to authentication entry point
```

- Se logger ensuite afin de vérifier qu'une fois authentifié nous avons bien accès à la page home.
- On s'aperçoit qu'une erreur se produit lors du clic sur le bouton Show Details sur la page Home. En effet, l'attribut de la session dans lequel le login de l'utilisateur est stocké n'est plus notre attribut personnel 'loginUser' mais un objet protégé de la session acegi. Modifier la méthode `getUserFromSession` de la classe `UserDWRImpl`.

Modification de l'implémentation UserDWRImpl

```
public User getUserFromSession(HttpServletRequest request) {
    SecurityContext context =
        (SecurityContext)request.getSession()
            .getAttribute("ACEGI_SECURITY_CONTEXT");
    String login = ((org.acegisecurity.userdetails.User)
        (context.getAuthentication().getPrincipal())).getUsername();
    if (null != login) {
        return this.getUser(login, request);
    }
    else {
        return null;
    }
}
```

- Le bouton Show Details devrait être désormais parfaitement opérationnel. On est allé, dans cette méthode, récupérer l'objet sécurisé acegi dans la session puis on a récupéré les paramètres d'authentification et l'objet `Principal` (l'utilisateur connecté) en particulier pour finalement accéder à son login stocké dans acegi sous l'appellation `userName`.

Grâce à cela, notre application est désormais sécurisée en terme d'accès aux urls. Un utilisateur ne peut donc accéder aux ressources protégées qu'à partir du moment où il est authentifié. On peut remarquer que cela permet de sécuriser l'ensemble des appels aux méthodes publiées par dwr puisque les urls correspondants à ces services (/dwr/**) font partie des urls protégées. Seuls les utilisateurs authentifiés pourront donc utiliser ces ressources et accéder grâce à elles à des méthodes du serveur.

Ce type de sécurisation est dans la majorité des cas bien suffisant pour les applications web classiques. Cependant, dans certains cas particuliers ou dans le cadre d'applications full Web 2.0 (une application = une et une seule url) il peut être utile voire nécessaire de sécuriser les appels aux méthodes du serveur. Une fois encore acegi permet de faire cela en se basant sur les principes de gestion des autorisations exposés plus haut. On doit cependant bien tenir compte de l'investissement non négligeable que cette mise en place peut représenter dans le cas de systèmes complexes (La sécurité a un coût).

VIII - Sécurisation des appels de méthodes

Selon la complexité de l'application et du métier, le niveau de sécurité que l'on souhaite atteindre, la sécurisation des appels de méthodes peut devenir un vrai casse-tête et même remettre en cause une partie de l'architecture logicielle ou tout du moins obliger la création d'une couche de sécurité dédiée composée uniquement de wrappers. En effet, si l'on souhaite sécuriser toutes les méthodes, y compris les simples readers, on se retrouve rapidement dans la situation où déterminer les droits d'accès à une méthode nécessite l'utilisation d'une autre méthode elle-même protégée, etc. Ce processus n'a de fin que si l'on sort de la couche sécurisée pour accéder directement aux couches inférieures. Cette situation n'étant pas acceptable d'un point de vue architectural, il est alors nécessaire de disposer d'une couche supérieure supplémentaire dédiée à la sécurité. Pour cette raison et bien d'autres encore, il est courant que seules les méthodes effectuant des opérations de suppression ou de modification ainsi que les méthodes très sensibles soient protégées de cette manière.

Dans le cadre de ce tutoriel nous n'allons pas se poser ce genre de question puisque nous allons simplement effectuer un contrôle afin de vérifier que seul un utilisateur a le droit de modifier son propre mot de passe. Il est cependant nécessaire de garder ces contraintes à l'esprit.

De manière générale les mécanismes de sécurité d'acegi sont gérés par l'utilisation de la POA (Programmation Orientée Aspect) mais cet aspect est totalement masqué pour nous par le framework. Dans le cadre de la sécurisation des appels de méthodes, il ne nous sera toujours pas nécessaire d'implémenter nous même ces mécanismes de POA. Cependant, nous devons dès à présent choisir quel sera le moteur de POA que nous utiliserons car cela influe considérablement sur la configuration nécessaire. Dans le cadre de ce tutoriel nous utiliserons le framework de l'AOP Alliance fourni nativement avec acegi. Noter qu'il est possible d'utiliser d'autres frameworks POA tels que AspectJ mais cela sort du cadre de ce tutoriel. Noter également que l'utilisation de ces framework dédiés implique souvent une connaissance plus approfondie des notions liées à la POA.

Pour mettre en place cette sécurisation des appels de méthodes, acegi - à l'instar de beaucoup de mécanismes Spring - fonctionne grâce à un mécanisme de *dynamic proxies*. Ainsi nous allons définir des services *virtuels* qui, grâce au proxy seront appelés en lieu et place des services réels et qui auront pour unique tâche d'effectuer la vérification des autorisations puis, le cas échéant, de passer la main au service métier, à la méthode protégée.

- Configurer le **proxy** :

Configuration du proxy de sécurité

```
<!-- ===== autoProxycreator ===== -->
<bean id="autoProxyCreator"
      class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
  <property name="interceptorNames">
    <list><value>tutoManagerSecurity</value></list>
  </property>
  <property name="beanNames">
    <list><value>userManager</value></list>
  </property>
</bean>
```

Cette configuration signifie que nous créons un nouveau service *virtuel* sous la forme d'un bean *userManagerSecurity* afin de protéger l'ensemble des appels aux méthodes du bean *userManager* défini par ailleurs (noter que la référence doit réellement pointer sur le nom d'un bean existant). On note que grâce aux attributs *list* on peut avoir plusieurs *Interceptor* et bien sûr sur plusieurs services sécurisés.

- Définir le **service sécurisé** :

Définition du service sécurisé

```
<!-- ===== tutoManagerSecurity ===== -->
<bean id="tutoManagerSecurity"
      class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
  <property name="validateConfigAttributes">
    <value>true</value>
  </property>
  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
  </property>
  <property name="accessDecisionManager">
    <ref bean="secureAccessDecisionManager"/>
  </property>
  <property name="objectDefinitionSource">
    <ref bean="secureObjectDefinitionSource"/>
  </property>
</bean>
```

On note le positionnement à true de l'attribut *validateConfigAttributes*. Cela signifie que le *MethodSecurityInterceptor* vérifiera au lancement de l'application que les attributs de configuration sont valides c'est-à-dire qu'au moins un voter les prendra en compte. Si ce n'est pas le cas une exception sera levée. On remarque la référence à l'*authenticationManager* habituelle ainsi que la référence à un nouvel *AccessDecisionManager* dédié et à un nouvel *ObjectDefinitionSource* permettant la définition des attributs de configuration. Il est à préciser qu'il existe de nombreuses configurations possibles et de nombreuses propriétés supplémentaires sur ce type de bean, plus ou moins complexes. Nous ne les passerons pas en revue dans le cadre de ce tutoriel. Se reporter à la documentation acegi pour plus de précisions.

- Définir le **AccessDecisionManager** et ses voters :

Définition du AccessDecisionManager

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

<bean id="himselfVoter" class="tuto.webssh.security.HimselfVoter">
  <property name="rolePrefix">
    <value>HIMSELF</value>
  </property>
</bean>
```

Dans ce bean, nous retrouvons la définition classique d'un *AccessDecisionManager* Unanimous avec, cette fois, deux voters : un *roleVoter* standard et un voter personnalisé. Pour que l'accès à une ressource configurée avec cet *accessDecisionManager* soit permis il faut donc que chacun de ces voters s'abstiennent ou vote en faveur de l'accès. Dans ce cas, le *HimselfVoter* est une classe personnelle qui ne votera qu'en présence de l'attribut de configuration 'HIMSELF' :

Définition du Voter personnalisé

```
package tuto.webssh.security;
```

Définition du Voter personnalisé

```
import org.acegisecurity.Authentication;
import org.acegisecurity.ConfigAttribute;
import org.acegisecurity.ConfigAttributeDefinition;
import org.acegisecurity.userdetails.UserDetails;
import org.acegisecurity.vote.RoleVoter;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.framework.ReflectiveMethodInvocation;

public class HimselfVoter extends RoleVoter {

    private Log logger = LogFactory.getLog(HimselfVoter.class);

    public boolean supports(ConfigAttribute configAttribute) {
        Boolean result = (configAttribute != null
            && configAttribute.getAttribute() != null
            && configAttribute.getAttribute().equals(
                this.getRolePrefix()));

        logger.info("support : result = "+result.toString());
        return result;
    }

    public boolean supports(Class _class) {
        logger.info("support class");
        return true;
    }

    public int vote(Authentication authentication, Object obj,
        ConfigAttributeDefinition configAttributeDefinition) {
        int result = ACCESS_ABSTAIN;
        String login = null;
        ReflectiveMethodInvocation methodInvocation = (ReflectiveMethodInvocation) obj;
        Object[] params = methodInvocation.getArguments();
        String methodName = methodInvocation.getMethod().getName();

        if (methodName != null && methodName.equals("changePassword")){
            login = (String)params[0];
        }
        String userName = ((UserDetails)authentication.getPrincipal()).getUsername();

        if (userName != null && login != null && userName.equals(login)){
            result = ACCESS_GRANTED;
            logger.info("Himself Vote: ACCESS GRANTED");
        }
        else {
            result = ACCESS_DENIED;
            logger.info("Himself Vote: ACCESS DENIED");
        }

        return result;
    }
}
```

Attardons nous quelques instants sur cette classe. Plusieurs méthodes sont à remarquer :

- La méthode **support (Class)** n'est à creuser que dans le cas où la participation du voter est liée à la classe appelante. Si on n'utilise pas cette fonctionnalité, il est indispensable de faire systématiquement renvoyer true à cette méthode de manière à pouvoir exécuter la seconde méthode support.
- La méthode **support (ConfigAttribute)** détermine si le voter doit participer au vote : en l'occurrence si l'attribut de configuration examiné à ce moment est exactement égal au préfixe configuré pour ce voter.

L'ensemble des attributs de configuration de la ressource protégée seront successivement examinés par cette méthode.

- La méthode vote ne sera appelée que si le voter doit participer à ce vote. C'est cette méthode qui votera réellement en faveur ou contre l'accès à la ressource. Dans le cas présent, cette méthode effectue une introspection afin de récupérer le nom de la méthode appelée et si il s'agit de la méthode `changePassword` vérifie que son premier paramètre (login) est bien égal au user connecté.

On remarque tout de suite que cette opération est loin d'être triviale et pourrait être bien plus élégante. On peut en effet imaginer regrouper différentes méthodes en fonction des critères qui déterminent leurs accès puis créer une annotation personnalisée sur ces méthodes permettant de récupérer la valeur ou l'index du paramètre nécessaire afin de ne pas lier le traitement au nom de la méthode lui-même. Cependant, la mise en place d'autorisations de ce type ne sera jamais ni triviale ni gratuite.

- Définir les attributs de configuration : Plusieurs stratégies sont possibles pour la définition de ces attributs. On peut en effet les spécifier dans le xml directement sous la forme : `package.Classe.methode=ATTRIBUT` qui permet de spécifier que cette méthode est protégée par les `AccessDecisionsManager` traitant des attributs de configuration 'ATTRIBUT'. Une autre solution est d'utiliser les annotations Java5 et de configurer les attributs au niveau de chaque méthode sécurisée plutôt que de manière centrale dans le xml. C'est cette deuxième méthode que nous allons exposer ici car elle me paraît à la fois la plus élégante, la plus lisible et la plus maintenable. Cela se fait par une référence à la classe `SecurityAnnotationAttributes`.

Définition des attributs de configuration

```
<!-- ===== objectDefinitionSource ===== -->
<bean id="attributes" class="org.acegisecurity.annotation.SecurityAnnotationAttributes"/>
<bean id="secureObjectDefinitionSource"
class="org.acegisecurity.intercept.method.MethodDefinitionAttributes">
  <property name="attributes"><ref local="attributes"/></property>
</bean>
```

- On va ensuite modifier la classe java sécurisée (celle mappée par le bean `userManager`) afin de configurer les attributs sur la méthode qui nous intéresse. Noter que l'on utilise ici des annotations Java5 mais qu'il est possible d'utiliser l'API Jakarta Commons Attributes en changeant toutefois la configuration xml. Toutefois, il est préférable d'utiliser les annotations Java5 standard dans la mesure du possible :

Sécurisation de la méthode `changePassword`

```
package tuto.webssh.service;

import org.acegisecurity.annotation.Secured;
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

import tuto.webssh.domain.model.User;

/**
 * This interface publishes business features to handler users
 * @author bmeurant
 */
@Transactional (readOnly=true, propagation=Propagation.REQUIRED)
public interface UserManager {

    /**
     * Check if the login exists and if the password is correct.
     * @param login : user login
     * @param password : user password
     * @return true if the login exists and if the password is correct.
     * Otherwise, return false.
     */
}
```

Sécurisation de la méthode changePassword

```

public boolean checkLogin (String login, String password);

/**
 * Return a User object from a given login.
 * @param login : user login
 * @return the corresponding user object.
 */
public User getUser(String login);

/**
 * Change the password to 'password' for the given login
 * @param login : user login
 * @param password : user new password
 * @return the new User object
 */
@Secured({"ROLE_AUTH", "HIMSELF"})
@Transactional (readOnly=false)
public User changePassword (String login, String password);
}
    
```

On spécifie ici que la méthode `changePassword` est sécurisée avec les attributs de configuration 'ROLE_AUTH' et 'HIMSELF' : L'ensemble des votes configurés pour participer aux votes faisant intervenir de tels attributs se prononcera donc sur cette autorisation. A noter que pour que cette nouvelle classe compile, il est impératif d'ajouter au classpath le jar `acegi-security-tiger-1.0.4.jar` fournit dans la distribution `acegi` standard (*tiger* étant le petit nom de Java5).

- Relancer le serveur, se connecter en tant que 'test' puis aller sur la page <http://localhost:8080/BlankApplicationAcegi/dwr/test/UserDWR>. Exécuter la méthode `changePassword` avec les paramètres 'test', 'test', 'test2'. La méthode s'exécute correctement et un objet javascript est affiché à l'écran. Les logs affichent : Himself Vote: ACCESS GRANTED.
- Exécuter la méthode `changePassword` avec les paramètres 'test2', 'test2', 'test3' après avoir ajouté un nouvel utilisateur 'test2', 'test2' en BDD. Une erreur survient et le message suivant apparaît. Les logs affichent : Himself Vote: ACCESS DENIED.



Figure4 : Interdiction d'exécution

IX - Conclusion

Ce tutoriel nous a permis de mettre en oeuvre les notions de base d'acegi security et d'explorer brièvement certains aspects avancés de sécurisation des appels de méthodes. Les possibilités d'acegi sont loin d'avoir été toutes explorées mais plus la configuration est fine et les fonctions avancées plus la complexité et le coût de maîtrise et de mise en #uvre seront élevés. Il reste alors à chacun à évaluer ses réels besoins de sécurité et le prix qu'il est prêt à payer en terme de performances, complexité, etc. Il est en effet possible de sécuriser les objets du domaine, de se greffer sur un CAS ou un SSO, d'effectuer des connexion et récupérations de droits automatiques depuis un LDAP, une BDD, etc.

Cependant, dans la plupart des cas, les simples fonctionnalités d'authentification et de sécurisation des urls sont bien suffisantes et offrent déjà un niveau de sécurité et de fonctionnalité bien supérieur à celui proposé par le standard J2EE de manière native.

En conclusion, Acegi Security permet de mettre facilement en place une couche de sécurité robuste et pérenne. Les aspects authentification et sécurisation des urls peuvent se configurer très simplement de manière à rendre un service standard, suffisant dans 90% des cas. Mais la grande force d'acegi est qu'il semble n'avoir aucune limite en terme de complexité. Ainsi il est possible d'effectuer des contrôles d'autorisations beaucoup plus évoluées à base de plusieurs voters. La contrepartie de cette grande flexibilité est que les divers mécanismes à mettre en #uvre ainsi que la configuration associés peuvent devenir rapidement assez complexes.

J'invite donc tous les développeurs travaillant sur une application spring à se jeter sans hésiter dans la mise en place des mécanismes les plus simple de sécurisation. Libre à eux d'aller plus loin en fonction de leurs besoins et du temps qu'ils sont disposés à y passer.

X - Liens

Quelques liens utiles :

- Article de ego sur acegi : <http://ego.developpez.com/acegi/acegi.pdf>
- Un chapitre de livre entier sur le sujet :
http://searchsoftwarequality.techtarget.com/searchAppSecurity/downloads/Spring_Framework_ch10.pdf
- Site officiel d'acegi : <http://www.acegisecurity.org/>
- Un article IBM : <http://www.ibm.com/developerworks/java/library/j-acegi1/>
- Article Wikipedia sur acegi : [http://en.wikipedia.org/wiki/Acegi_security_framework_\(Java\)](http://en.wikipedia.org/wiki/Acegi_security_framework_(Java))

XI - Remerciements

Merci à **Hikage** et **ego** pour la relecture technique.

Merci à **jeepnc** pour la relecture orthographique.

Ce document a été publié avec l'autorisation de la société qui m'emploie : **Atos Worldline**.

XII - Dans la même série ...

Ce tutoriel est le deuxième d'une série de 3. Accéder aux autres tutoriels de cette série :

- **Premier volet : Premier projet avec Tapestry5, Spring et Hibernate**
- **Deuxième volet : Intégration simple et élégante d'AJAX avec DWR**
- **Troisième volet : Sécuriser vos applications Web avec Spring acegi security -> En Relecture**

