

Sécuriser une application avec

ACEGI SECURITY SYSTEM

Auteur : Erik Gollot (septembre 2006)



Acegi
**SECURITY
SYSTEM**
FOR SPRING

L'objectif de cet article est de vous présenter le framework « ACEGI Security system » qui permet de sécuriser vos applications Spring. Nous verrons dans un premier temps ce qu'est exactement ce framework, pourquoi il a été créé et enfin nous verrons, par l'exemple, différentes fonctionnalités.

Qu'est ce qu' ACEGI SECURITY SYSTEM ?

Introduction

Le framework ACEGI est un framework de sécurité. Eh ! pas mal comme information non ? Bon, soyons un peu plus précis; ACEGI est un framework qui va vous permettre de gérer 2 grandes problématiques liées à la sécurité applicative :

1. Qui es-tu toi qui parles à mon application ? Ça c'est l'**authentification**
2. Qu'as-tu le droit de faire avec mon application ? Ça c'est l'**autorisation**

C'est cool mais à quoi cela sert-il puisque la sécurité est déjà quelque chose intégré à Java EE au travers de la spécification sur les [servlets](#) et sur les [EJB](#) ?

Et bien le problème est que ces spécifications s'appuient sur la notion de rôle mais que la manière d'associer un rôle avec un « principal », on va dire un « nom d'utilisateur » pour faire simple, n'est pas standardisée. La conséquence est simple, chaque serveur d'application propose ses propres extensions pour réaliser cette correspondance. Je ne prendrai qu'un exemple simple avec le fichier tomcat-users.xml de Tomcat qui, comme son nom semble l'indiquer, est spécifique à Tomcat.

Alors, bon, ce n'est pas non plus tous les quatre matins que l'on change de serveur d'application, je vous l'accorde. Cependant, il n'est pas rare que vous ayez, dans votre entreprise ou pour des clients, à travailler avec différents serveurs d'application. D'autre part, on va le voir au travers des différents exemples, ACEGI est totalement intégré à Spring et bénéficie donc du mécanisme d'IoC et vous permet in fine d'être homogène dans la manière de bâtir et de sécuriser votre application.

Enfin, on peut aussi préciser qu'en fonction des options de sécurité choisies, vous allez pouvoir fournir une application sécurisée auto-suffisante pour n'importe quel serveur d'application puisque toutes les informations de sécurité, y compris les bibliothèques ACEGI, seront packagées avec votre application sous forme de .WAR ou .EAR.

Les fonctionnalités en un clin d'oeil

Côté authentification

Les modes d'authentification supportés sont les suivants :

- HTTP BASIC
- HTTP Digest
- HTTP X.509 (échange de certificat)
- LDAP
- Form-based (comme dans le cas des servlets)
- SiteMinder (de Computer Associates)
- JA-SIG Central Authentication Service (CAS – outil de SSO open source)
- Propagation du contexte d'authentification pour RMI et HttpInvoker de Spring
- Possibilité de se rappeler d'une précédente authentification (« remember-me »)
- Authentification anonyme

- « Run-as » permettant de changer l'identité de l'appelant quand un serveur appelle un autre serveur
- Java Authentication and Authorization (JAAS)
- Votre propre système d'authentification (par extension)

Côté autorisation

Les éléments d'une application pouvant être sécurisés sont :

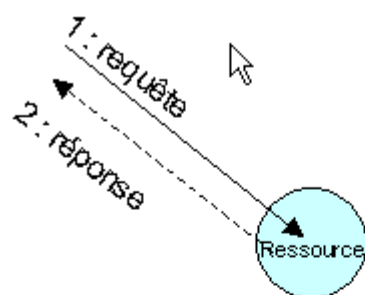
- Les URLs de votre site, c'est à dire les pages et les servlets
- Les méthodes de vos beans. Là, on dépasse ce que sait faire en standard Java EE puisque ce n'est possible qu'avec des EJBs
- Les objets eux mêmes. Là on dépasse encore plus ce que propose Java EE en standard

Je n'ai bien entendu pas listé toutes les possibilités et vous vous reporterez à la [documentation officielle](#) pour tous les détails.

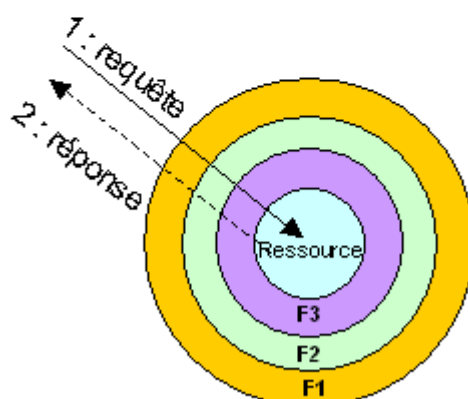
Les grands principes

Le filtrage des accès

Accès à une ressource non sécurisée



Accès à une ressource sécurisée



Une série de filtres (F1, F2,...) sont intercalés entre l'appelant et la ressource

Le principe mis en place par ACEGI pour sécuriser une ressource est relativement simple, une série de « filtres » sont interposés entre l'appelant et la ressource elle-même. Ces différents filtres ont chacun un rôle précis dans la chaîne de sécurisation. Nous allons voir aussi que l'on peut mettre ou ne pas mettre certains filtres en fonction des options que l'on choisit pour sécuriser la ressource mais aussi en fonction des besoins qui peuvent s'imposer.

ACEGI utilise la notion de « filter » définie dans la spécification sur les servlets (fichier web.xml) pour positionner ses propres filtres spécialisés dans la sécurité et les aspects (Spring AOP ou AspectJ) pour sécuriser les appels de méthodes ou les objets.

Où sont les informations sur la sécurité ?

Un objet est au centre de tout le dispositif, le `SecurityContextHolder`. Cet objet contient toutes les informations nécessaires à la gestion de la sécurité. Comme par défaut cet objet utilise un `ThreadLocal`, il est accessible à tous les objets d'un même thread même si ces objets n'ont pas d'opérations spécifiquement dédiées au passage du contexte de sécurité. Certains cas d'usage particuliers de `ThreadLocal` sont exposés dans la documentation ACEGI.

Les informations sur « Qui me parle ? » (le « principal ») sont stockées dans un objet `Authentication`. Le code suivant montre comment récupérer le nom de l'utilisateur :

```
Object obj = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
if (obj instanceof UserDetails) {
    String username = ((UserDetails)obj).getUsername();
} else {
    String username = obj.toString();
}
```

NB : je passe sous silence les explications sur la classe `UserDetails`, nous verrons cela dans un des exemples.

Présentation de l'exemple

Comme vous avez pu le voir, les fonctionnalités d'ACEGI sont extrêmement nombreuses. Je ne vais donc pas me lancer dans un exemple implémentant toutes ces fonctionnalités car je suis un fainéant et surtout parce que je ne suis pas un expert en sécurité et que donc je ne maîtrise pas toutes les technologies proposées. Nous allons donc nous restreindre aux éléments suivants :

Définition d'un WebService avec HttpInvoker avec :

- Authentification de type HTTP Basic
- Autorisations de certaines méthodes pour les rôles « USER » et « ADMIN »
- Autorisations de certaines méthodes pour le rôle « ADMIN » uniquement
- Stockage des « login, mot de passe, rôles » dans un fichier XML (option 1)
- Stockage des « login, mot de passe, rôles » dans une base de données (option 2)

Définition d'un client Java simple avec :

- Appel du WebService précédent
- Mise en oeuvre d'une authentification de type HTTP Basic au sein d'une client non Web

Définition d'un WebService avec HttpInvoker

Commençons d'abord par définir un WebService avec l'aide de `HttpInvoker` offert par Spring. Pour cela, nous allons définir un service qui permet de rechercher des « News » dans une base de données relationnelles. Pour faire une application qui fonctionne réellement, je vous propose d'utiliser Hibernate pour le mapping objet-relationnel ainsi qu'une base MySQL. C'est un peu beaucoup pour un si petit exemple mais au moins, on aura un contexte proche de ce que vous pourrez rencontrer dans vos projets. Je passerai cependant sous silence les explications sur Hibernate car ce n'est pas fondamental pour nos histoires de sécurité.

L'interface du service

Bon, on commence par un objet `News` tout simple avec un titre et une description plus un id et un numéro de version. Ces 2 derniers attributs étant gérés par Hibernate.

```
package com.devcom.acegi.news;

import java.io.Serializable;

public class News implements Serializable {

    private long id;
    private long version;
    private String title;
    private String description;
}
```

```

public String getDescription() {
    return description;
}
public void setDescription(String text) {
    this.description = text;
}
public String getTitle() {
    return title;
}
public void setTitle(String title) {
    this.title = title;
}
public long getId() {
    return id;
}
public void setId(long id) {
    this.id = id;
}
}

```

On définit ensuite une interface définissant les services offerts au dessus de nos News, l'interface NewsManager.

```

package com.devcom.acegi.news;

import java.util.Collection;

public interface NewsManager {
    public Collection getAllNews();
    public void createNews(News n);
    public void deleteNews(News n);
}

```

Ces 2 classes sont packagées dans le jar **news.jar**.

L'implémentation POJO du service

L'implémentation de notre service est réalisée au travers de 2 classes, une classe NewsDAO chargée des requêtes vers la base de données et la classe NewsManagerImpl chargée d'implémenter l'interface NewsManager.

```

package com.devcom.acegi.news.dao;

import java.util.Collection;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import com.devcom.acegi.news.News;

public class NewsDAOImpl extends HibernateDaoSupport implements NewsDAO {

    public Collection findAll() {
        return getHibernateTemplate().find("from News");
    }

    public News findByPrimaryKey(long key) {
        return (News) getHibernateTemplate().load(News.class, new Long(key));
    }

    public void save(News n) {
        getHibernateTemplate().save(n);
    }

    public void delete(News n) {
        getHibernateTemplate().delete(n);
    }
}

```

```

package com.devcom.acegi.news.server;

import java.util.Collection;

import com.devcom.acegi.news.News;
import com.devcom.acegi.news.NewsManager;
import com.devcom.acegi.news.dao.NewsDAO;

```

```

public class NewsManagerImpl implements NewsManager {

    private NewsDAO dao;

    public Collection getAllNews() {
        return getDao().findAll();
    }
    public void createNews(News n) {
        getDao().save(n);
    }
    public void deleteNews(News n) {
        getDao().delete(n);
    }
    public NewsDAO getDao() {
        return dao;
    }
    public void setDao(NewsDAO dao) {
        this.dao = dao;
    }
}

```

Pour faire communiquer notre service avec la classe DAO et notre base de données MySQL, on définit le fichier de configuration Spring applicationContext.xml suivant :

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <!-- Definition de la source de donnees sous MySQL -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-
method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql:///acegi"/>
        <property name="username" value="devcom"/>
        <property name="password" value="devcom"/>
    </bean>

    <!-- Definition des fichiers de mapping O/R Hibernate -->
    <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="mappingResources">
            <list>
                <value>com/devcom/acegi/news/News.hbm.xml</value>
            </list>
        </property>
        <property name="hibernateProperties">
            <props>
                <prop
key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
                <prop key="hibernate.show_sql">true</prop>
            </props>
        </property>
    </bean>

    <!-- Definition du transaction manager a utiliser pour rendre les methodes du service
transactionnelles -->
    <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <!-- Definition de notre implementation du DAO utilisant Hibernate -->
    <bean id="newsDAO" class="com.devcom.acegi.news.dao.NewsDAOImpl">
        <property name="sessionFactory" ref="sessionFactory"/>
    </bean>

    <!-- Definition du service non encore transactionnel -->
    <bean id="newsManagerTarget" class="com.devcom.acegi.news.server.NewsManagerImpl">
        <property name="dao" ref="newsDAO"/>
    </bean>

    <!-- Definition du service avec ses caracteristiques transactionnelles -->
    <bean id="newsManager"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
        <property name="transactionManager" ref="transactionManager"/>
        <property name="target" ref="newsManagerTarget"/>
        <property name="transactionAttributes">

```

```

        <props>
            <prop key="*">PROPAGATION_REQUIRED</prop>
        </props>
        </property>
    </bean>
</beans>

```

Enfin, afin de rendre notre service « newsManager » accessible sous la forme d'un service Web, nous avons besoin de créer une « Web application » (newsserverws.war) pour laquelle, outre toutes les bibliothèques nécessaires, il faut définir les fichiers web.xml et httpinvoker-servlet.xml suivants :

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <context-param>
        <param-name>contextConfiguration</param-name>
        <param-value>/WEB-INF/applicationContext.xml</param-value>
    </context-param>
    <servlet>
        <servlet-name>context</servlet-name>
        <servlet-class>org.springframework.web.context.ContextLoaderServlet</servlet-
class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet>
        <servlet-name>httpinvoker</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
        <load-on-startup>2</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>httpinvoker</servlet-name>
        <url-pattern>/ws/*</url-pattern>
    </servlet-mapping>
</web-app>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <!-- Le service sera deduit du nom de l'URL -->
    <bean id="defaultHandlerMapping"
        class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping" />
    <!-- Exposition du newsManager sous la forme d'un service Web -->
    <bean name="/NewsManagerService.do"
class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
        <property name="service" ref="newsManager"/>
        <property name="serviceInterface" value="com.devcom.acegi.news.NewsManager"/>
    </bean>
</beans>

```

A ce stade, notre « Webservice » est accessible sans aucune restriction.

La classe suivante NewsClient et le fichier de configuration Spring applicationContext.xml associé permettent de tester le service.

```

package com.devcom.acegi.news.client;

import java.util.Collection;
import java.util.Iterator;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.devcom.acegi.news.News;
import com.devcom.acegi.news.NewsManager;

public class NewsClient {

    private NewsManager manager;

    public static void main(String[] args) {
        ClassPathXmlApplicationContext factory = null;

```

```

        factory = new ClassPathXmlApplicationContext(
            new String[] {"applicationContext.xml"});

        NewsClient client = (NewsClient)factory.getBean("newsClient");
        client.showAllNews();

    }

    public NewsManager getManager() {
        return manager;
    }

    public void setManager(NewsManager manager) {
        this.manager = manager;
    }

    public void showAllNews() {
        Collection all = getManager().getAllNews();
        Iterator i = all.iterator();
        News n;
        while (i.hasNext()) {
            n = (News)i.next();
            System.out.println("Id : " + n.getId());
            System.out.println("Titre " + n.getTitle());
            System.out.println("Description " + n.getDescription());
            System.out.println("");
        }
    }
}

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

    <bean id="newsClient" class="com.devcom.acegi.news.client.NewsClient">
        <property name="manager" ref="newsManager"/>
    </bean>

    <bean id="newsManager"
        class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
        <property name="serviceUrl"
value="http://localhost:8080/newsserverws/ws/NewsManagerService.do"/>
        <property name="serviceInterface" value="com.devcom.acegi.news.NewsManager"/>
    </bean>
</beans>

```

Une implémentation sécurisée

Voyons maintenant comment sécuriser les accès à notre Webservice.

Mise en place du filtre

Tout d'abord, nous devons mettre en place la logique de filtres d'ACEGI. Pour cela, il faut mettre à jour le fichier web.xml de notre WebApp en y définissant un filtre au sens servlet. Ce filtre va charger sa propre configuration depuis les fichiers de configuration Spring.

```

<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <context-param>
        <param-name>contextConfiguration</param-name>
        <param-value>
            /WEB-INF/applicationContext.xml
        </param-value>
    </context-param>
    <filter>
        <filter-name>Acegi Filter Chain Proxy</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-mapping>
  <filter-name>Acegi Filter Chain Proxy</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>

  <listener>
    <listener-
class>org.springframework.web.context.ContextLoaderListener</listener-class>
    </listener>
    <listener>
      <listener-class>org.springframework.web.util.Log4jConfigListener</listener-
class>
    </listener>
    <servlet>
      <servlet-name>httpinvoker</servlet-name>
      <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-
class>
      <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
      <servlet-name>httpinvoker</servlet-name>
      <url-pattern>/ws/*</url-pattern>
    </servlet-mapping>
</web-app>

```

Vous noterez la déclaration en gras ci-dessus et le changement du mode de chargement des fichiers de configuration Spring. On passe maintenant par l'usage d'un listener et non d'une servlet, ceci à cause de l'usage des filtres et de l'ordre de démarrage des filtres et servlets par le container. En clair, si vous conservez la servlet « ContextLoaderServlet » de la version précédente de web.xml, les filtres ne trouvent pas leurs fichiers de configuration.

Déclaration de la chaîne de sécurisation

Dans notre fichier de configuration Spring, nous allons maintenant déclarer les filtres ACEGI chargés de mettre en oeuvre notre stratégie de sécurisation. Pour faire propre, il faudrait créer plusieurs fichiers de configuration mais pour tout avoir sous les yeux, je vous propose de tout mettre dans un seul fichier.

```

<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
  <property name="filterInvocationDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
    </value>
  </property>
  <property name="doFilter">
    <value>
      /**/*.do=httpSessionContextIntegrationFilter,basicProcessingFilter,filterInvocationIntercep
tor
    </value>
  </property>
</bean>

```

La stratégie de sécurisation est définie au travers de la propriété `filterInvocationDefinitionSource`. Cette propriété possède une syntaxe particulière interprétée par la classe `FilterChainProxy`. Regardons les différents éléments un par un :

CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON

Cette instruction pré-définie permet de mettre en minuscule les URLs demandées par un client avant d'appliquer les règles de sécurisation. Cela permet d'éviter des erreurs de « case » lors de la définition ultérieure d'expressions régulières. Cette instruction est optionnelle mais comme tous les exemples ACEGI l'utilisent, je fais le mouton ! :-)

PATTERN_TYPE_APACHE_ANT

Cette instruction pré-définie permet d'expliquer que les expressions régulières définies par la suite suivent la syntaxe des expressions régulières de l'outil ANT. Cette syntaxe est plus simple que la syntaxe des expressions régulières Java classique qui est la syntaxe par défaut si on ne spécifie pas le type ANT.

//*.do=httpSessionContextIntegrationFilter,basicProcessingFilter,filterInvocationInterceptor**

Cette dernière ligne est la plus intéressante car c'est elle qui définit effectivement les filtres appliqués aux requêtes client. Cette ligne est composée de 2 parties.

La première partie « `/**/.do` » indique sur quelles URIs la sécurisation va opérer. Ici, on dit que toutes les URI se terminant par `.do` quelque soit la longueur de l'URI vont être sécurisées.

La seconde partie est une **liste ordonnée** de filtres appelés lors de la réception d'une requête et dont on va voir la déclaration précise par la suite :

`httpSessionContextIntegrationFilter` : ce filtre permet de stocker les informations de sécurité dans la session HTTP pour ne pas à avoir à redemander ces informations une fois la personne authentifiée. Dans le cas particulier de notre Webservice, ce filtre n'est pas utile car la session ne vit que pendant le temps de l'appel du Webservice. J'ai quand même voulu vous présenter ce filtre car dans la majorité des applications Web, ce filtre doit être déclaré en premier. Dans notre cas, il fait bien son boulot mais cela ne sert pas à grand chose.

`basicProcessingFilter` : ce filtre prend en charge de mode d'authentification de type « HTTP BASIC », c'est ce que l'on veut faire dans notre exemple.

`filterInvocationInterceptor` : ce filtre sert à la définition des droits sur notre Webservice (autorisation)

On peut déjà noter que si le filtre `basicProcessingFilter` n'autorise pas le client, le filtre `filterInvocationInterceptor` ne sera pas appelé.

Déclaration des sources pour l'authentification

Pour que les filtres puissent fonctionner, il faut maintenant déclarer les sources où se trouvent les informations de type login/password ainsi que les groupes d'appartenance de chaque login.

Pour cela, ACEGI offre la possibilité de définir un ensemble de sources, les « providers », qui vont être interrogés pour rechercher le « principal » (le login). On va donc pouvoir mixer plusieurs sources comme une source de type fichier, une source de type base de données, une source de type LDAP, etc... Dans notre exemple, nous allons commencer avec une source de type fichier.

Cette déclaration se fait en définissant un « `authenticationManager` ».

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

Nous avons donc ici un seul « provider » pour l'authentification. Pour faire simple dans un premier temps, nous allons utiliser une classe offerte par ACEGI qui permet la définition d'une « base de données » en mémoire, un « `InMemoryDAO` ». La déclaration est faite en 2 temps, la première déclaration définit un provider de type « DAO » (accès à une base de données) puis, comme ACEGI est configurable et permet de définir son propre « DAO provider », on va dire que le DAO est en fait un « `InMemoryDAO` ».

```
<bean id="daoAuthenticationProvider"
class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userService"><ref local="inMemoryDaoImpl"/></property>
</bean>

<bean id="inMemoryDaoImpl" class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      user1=password,ROLE_USER
      user2=password,ROLE_USER
      user3=password,disabled,ROLE_ADMIN
      admin=admin,ROLE_USER,ROLE_ADMIN
    </value>
  </property>
</bean>
```

La propriété « `userMap` » du `InMemoryDaoImpl` permet de définir une liste de login/password/rôles. On peut même, en seconde position après le signe « = », mettre le mot clé « `disabled` » pour interdire un login existant. Le nom des rôles est libre mais l'utilisation de la

syntaxe **ROLE_** permet d'utiliser les capacités d'autres classes fournies par ACEGI pour la gestion des autorisations.

Pour être plus propre et permettre de changer la liste des login/password/rôles, il est préférable de configurer la « `inMemoryDAOImpl` » en référençant un fichier de propriétés comme suit :

```
<bean id="inMemoryDaoImpl" class="org.acegisecurity.userdetails.memory.InMemoryDaoImpl">
  <property name="userProperties">
    <bean class="org.springframework.beans.factory.config.PropertiesFactoryBean">
      <property name="location" value="/WEB-INF/users.properties"/>
    </bean>
  </property>
</bean>
```

Le fichier « `users.properties` » contient les mêmes informations que précédemment mais il permet une modification de ces informations lors du déploiement.

```
user1=password,ROLE_USER
user2=password,ROLE_USER
user3=password,disabled,ROLE_ADMIN
admin=admin,ROLE_USER,ROLE_ADMIN
```

Pas mal, mais le problème c'est que les mots de passe sont en clair dans ce fichier. Heureusement, ACEGI permet de gérer l'encoding des mots de passe en utilisant MD5, SHA ou le texte en clair comme on l'a fait jusqu'à présent.

Afin de rendre illisible les mots de passe, je vous propose donc d'utiliser SHA pour crypter les mots de passe.

```
<bean id="daoAuthenticationProvider"
class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userService"><ref local="inMemoryDaoImpl"/></property>
  <property name="passwordEncoder" ref="shaEncoder"/>
</bean>

<bean id="shaEncoder" class="org.acegisecurity.providers.encoding.ShaPasswordEncoder"/>
```

Pour encoder les mots de passe précédents, j'ai utilisé la classe `ShaPasswordEncoder` pour faire un petit exécutable auquel je passe le mot de passe en clair et il me retourne le mot de passe encodé.

```
package com.devcom.acegi.encoding;

import org.acegisecurity.providers.encoding.ShaPasswordEncoder;

public class SSHA {

    public static void main(String[] args) {
        ShaPasswordEncoder encoder = new ShaPasswordEncoder();
        String encodedPassword = encoder.encodePassword(args[0],null);
        System.out.println("{SHA}"+encodedPassword);
    }
}
```

Le fichier « `users.properties` » est donc le suivant maintenant :

```
user1=5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8,ROLE_USER
user2=5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8,ROLE_USER
user3=5baa61e4c9b93f3f0682250b6cf8331b7ee68fd8,disabled,ROLE_ADMIN
admin=d033e22ae348aeb5660fc2140aec35850c4da997,ROLE_USER,ROLE_ADMIN
```

Bon, c'est pas mal tout cela mais on peut faire encore mieux en stockant les informations dans une base de données. Là vous avez la possibilité d'utiliser le schéma de base proposé par ACEGI, c'est plus simple bien sûr, ou bien vous pouvez utiliser votre propre schéma. Dans ce dernier cas, il y a plus de boulot car il va falloir que vous preniez en charge vous même les requêtes à la base et la création des objets `UserDetails` attendus par ACEGI.

Regardons comment nous pouvons utiliser le schéma par défaut proposé par ACEGI.

Pour cela, on ne va pas remplacer notre « `inMemoryDAO` » mais on va simplement ajouter un « `provider` ».

```

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

<bean id="dbAuthenticationProvider"
class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">
  <property name="userDetailsService"><ref local="dbDaoImpl"/></property>
  <property name="passwordEncoder" ref="shaEncoder"/>
</bean>

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

```

J'ai donc ajouté le « dbAuthenticationProvider » que j'ai associé à un « JdbcDAOImpl ». La dataSource étant celle utilisée pour définir ma connexion vers ma base de données contenant les News (j'aurais bien entendu pu en définir une autre).

Les tables à créer sont les suivantes :

```

CREATE TABLE `users` (
  `username` varchar(255) NOT NULL default '',
  `password` varchar(255) NOT NULL default '',
  `enabled` varchar(10) NOT NULL default 'true'
);
CREATE TABLE `authorities` (
  `username` varchar(255) NOT NULL default '',
  `authority` varchar(255) NOT NULL default ''
);

```

Pour l'exemple, j'ai enlevé la ligne « admin » du fichier « users.properties » et j'ai exécuté les ordres suivants pour mettre le compte « admin » dans ma base mySQL.

```

INSERT INTO `users` VALUES ('admin', 'd033e22ae348aeb5660fc2140aec35850c4da997', 'true');
INSERT INTO `authorities` VALUES ('admin', 'ROLE_USER');
INSERT INTO `authorities` VALUES ('admin', 'ROLE_ADMIN');

```

Enfin, tout ceci doit être associé à notre fameux basicProcessingFilter, vous vous rappelez, c'est le filtre qui se charge de gérer l'authentification de type « HTTP BASIC ».

```

<bean id="basicProcessingFilter"
class="org.acegisecurity.ui.basicauth.BasicProcessingFilter">
  <property name="authenticationManager"><ref
local="authenticationManager"/></property>
  <property name="authenticationEntryPoint"><ref
local="basicProcessingFilterEntryPoint"/></property>
</bean>

<bean id="basicProcessingFilterEntryPoint"
class="org.acegisecurity.ui.basicauth.BasicProcessingFilterEntryPoint">
  <property name="realmName"><value>News Realm</value></property>
</bean>

```

La propriété « basicProcessingFilterEntryPoint » permet de définir comment vont être gérées les authentifications qui échouent. L'utilisation de la classe BasicProcessingFilterEntryPoint permet la génération d'une erreur HTTP 401 qui permet au client de recommencer le processus d'authentification.

Au fait, la déclaration de notre filtre qui stocke les informations d'authentification dans la session HTTP est ultra simple (pour une fois !) :

```

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

```

Déclaration des autorisations

Bon, ça y est, on en a fini avec l'authentification, on va maintenant pouvoir définir qui peut faire quoi avec notre Webservice.

Là encore, ACEGI est extrêmement ouvert et complet car il permet de définir non seulement des droits en utilisant simplement la notion de rôle mais on peut en fait utiliser n'importe quelle propriété de notre objet `UserDetails`. Cet objet que j'ai rapidement mentionné dans un chapitre précédent est l'objet qui contient les informations sur « Qui me parle ? ». Vous vous souvenez, on a vu comment récupérer le « `username` » en début d'article. Cet objet peut en fait être personnalisé lorsque vous avez plusieurs informations à stocker parce que ces informations vous sont utiles pour savoir qui peut faire quoi. Vous pouvez par exemple, stocker le service de l'entreprise dans laquelle la personne qui parle à votre application travaille et utiliser le nom du service pour l'autoriser ou non à utiliser certaines ressources.

Dans notre exemple, je ne me suis pas amusé à définir mon propre « `UserDetails` », j'utilise donc l'objet par défaut proposé par ACEGI.

Comment ça fonctionne les autorisations ?

Avant de rentrer dans le vif du sujet, regardons un peu la philosophie d'ACEGI au sujet des autorisations. Comme pour l'authentification, on passe pas la notion de filtre qui intercepte les appels et qui va ou non autoriser l'accès à la ressource. On parle d'ailleurs plutôt d'intercepteurs que de filtre dans le cas des autorisations car pour le positionnement des droits sur les méthodes et les objets, ACEGI utilise, entre autre, la notion d'intercepteur de Spring AOP.

Nos intercepteurs vont utiliser un « `accessDecisionManager` » qui est chargé de décider si on peut ou non accéder à une ressource.

ACEGI propose plusieurs « `accessDecisionManagers` »; ces managers vont utiliser une liste de « `voters` » qui sont des objets qui vont, en fonction de différents critères, accepter ou refuser l'accès à une ressource. Ces « `voters` » jouent un peu le rôle que nous jouons nous même quand on va voter pour une élection présidentielle ou autre. Notre vote ne décide pas in fine qui sera élu mais, en fonction des types d'élection, c'est la majorité absolue ou relative, par exemple, qui fait que telle ou telle personne est élue. Avec ACEGI, c'est un peu pareil, les « `voters` » décident à leur niveau si la ressource est ou non accessible mais c'est le manager qui décide. On a donc par exemple en standard un « `AffirmativeBased` » manager qui donne l'accès à la ressource si au moins un « voter » dit ok; un « `ConsensusBased` » manager qui donne l'accès à la ressource si la majorité des « `voters` » disent ok; un « `UnanimousBased` » manager qui donne l'accès à la ressource si tous les « `voters` » disent ok.

Dans chaque cas, le « voter » peut dire : ok, nok ou s'abstenir. L'abstention est prise en compte par les managers présentés précédemment.

Maintenant, au niveau des « `voters` », ACEGI propose par défaut 2 sortes de « `voters` » :

- `RoleVoter` : il s'appuie uniquement sur la notion de rôle et travaille par défaut avec les rôles dont le nom est préfixé par `ROLE_`.
- `BasicAclEntryVoter` : il permet de mettre en place des droits de type ACL : READ, WRITE, DELETE,...

Enfin, au niveau des intercepteurs, on va définir les droits que l'on veut positionner sur les différentes ressources au travers de la propriété « `objectDefinitionSource` ». Cette propriété est définie avec une syntaxe similaire à la syntaxe de notre liste de filtres « `FilterChainProxy` »

Voici ce que cela donne dans notre exemple :

```
<bean id="filterInvocationInterceptor"
class="org.acegisecurity.intercept.web.FilterSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="decisionManager"/>
  <property name="objectDefinitionSource">
    <value>
      CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
      PATTERN_TYPE_APACHE_ANT
      /**/*.do=ROLE_ADMIN
    </value>
  </property>
</bean>
<bean id="decisionManager" class="org.acegisecurity.vote.AffirmativeBased">
```

```

        <property name="allowIfAllAbstainDecisions" value="false"/>
        <property name="decisionVoters">
            <list>
                <bean class="org.acegisecurity.vote.RoleVoter"/>
            </list>
        </property>
    </bean>

```

Remarquons rapidement la propriété « allowIfAllAbstainDecisions » qui permet au « decisionManager » de décider de l'autorisation si tous les « voters » s'abstiennent.

L'expression « /**/*.do=ROLE_ADMIN » fait que seuls les utilisateurs ayant le rôle « ROLE_ADMIN » peuvent appeler notre Webservice. Avec cette syntaxe, le positionnement des permissions est relativement restreint puisque je ne définis des droits qu'au niveau du Webservice et non au niveau de ces différentes méthodes. C'est déjà un premier pas si l'on veut sécuriser les appels et n'autoriser l'usage du WeServices qu'à ceux qui « savent ».

Pour sécuriser les méthodes d'un objet, il va nous falloir passer par les mécanismes de l'AOP. Pour cela, ACEGI permet l'usage de Spring AOP ou d'AspectJ. Regardons l'usage de Spring AOP.

Sécuriser avec Spring AOP

Pour ceux qui ne connaissent pas l'AOP (Aspect Oriented Programming), et bien c'est pas grave. La seule chose que vous devez comprendre pour notre histoire de sécurité, c'est que nous allons pouvoir, au niveau de notre classe « NewsManager », intercaler un intercepteur, oui encore une fois ces foutus intercepteurs, qui va intercepter les appels au « NewsManager » (c'est normal, c'est le rôle d'un intercepteur !) et va tester les droits qui vont être positionnés maintenant sur les différentes méthodes de la classe.

Nous allons dans un premier temps supprimer notre « filterInvocationInterceptor » dans la liste du bean « filterChainProxy » ainsi que du fichier de configuration Spring.

```

<bean id="filterChainProxy" class="org.acegisecurity.util.FilterChainProxy">
    <property name="filterInvocationDefinitionSource">
        <value>
            CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
            PATTERN_TYPE_APACHE_ANT
            /**/*.do=httpSessionContextIntegrationFilter,basicProcessingFilter
        </value>
    </property>
</bean>

```

Ensuite, nous allons définir un intercepteur qui va définir les accès sur notre interface « NewsManager ». Oui, je dis interface car même si c'est la classe « NewsManagerImpl » qui est in fin appelée, on peut définir les droits sur l'interface que cette classe implémente.

```

<bean id="newsManagerSecurityInterceptor"
class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
    <property name="authenticationManager" ref="authenticationManager"/>
    <property name="accessDecisionManager" ref="decisionManager"/>
    <property name="objectDefinitionSource">
        <value>
            com.devcom.acegi.news.NewsManager.getAllNews=ROLE_USER,ROLE_ADMIN
            com.devcom.acegi.news.NewsManager.createNews=ROLE_ADMIN
            com.devcom.acegi.news.NewsManager.deleteNews=ROLE_ADMIN
        </value>
    </property>
</bean>

```

Notez la syntaxe des lignes de la propriété « objectDefinitionSource », il faut mettre le « Full Qualified Name » de notre interface suivie de « .<nom de méthode> ».

Maintenant, il nous reste à appliquer cet intercepteur au bon endroit. Le bon endroit, c'est là où on a définit les propriétés transactionnelles de notre service, là où on utilise un « TransactionProxyFactoryBean ». Pour ce type de déclaration, on a la possibilité de définir des « pré-intercepteurs » qui vont donc être appelés avant l'appel effectif de l'objet dit « target » (notre NewsManagerImpl). Notre « NewsManager » va ainsi être protégé par notre « SecurityInterceptor ».

```

<bean id="newsManager"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
  <property name="transactionManager" ref="transactionManager"/>
  <property name="target" ref="newsManagerTarget"/>
  <property name="transactionAttributes">
  <props>
    <prop key="*">PROPAGATION_REQUIRED</prop>
  </props>
  </property>
  <property name="preInterceptors">
    <list>
      <ref bean="newsManagerSecurityInterceptor"/>
    </list>
  </property>
</bean>

```

Et voilà, l'affaire est dans le sac, on a un beau Webservice sécurisé. Il ne nous reste plus qu'à voir comment un client peut appeler ce service.

Définition d'un client Java

Pour ceux qui connaissent déjà un peu Spring, la définition d'un client faisant appel à un Webservice défini avec un HttpInvoker est relativement simple. Il suffit d'utiliser la classe « HttpInvokerProxyFactoryBean » et de lui fournir :

- L'URL du service
- L'interface métier implémentée par le service

```

<bean id="newsManager"
      class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
value="http://localhost:8080/newsserverws/ws/NewsManagerService.do"/>
  <property name="serviceInterface" value="com.devcom.acegi.news.NewsManager"/>
</bean>
<bean id="newsClient" class="com.devcom.acegi.news.client.NewsClient">
  <property name="manager" ref="newsManager"/>
</bean>

```

L'utilisation de ce bean se fait simplement comme suit :

```

public class NewsClient {

    private NewsManager manager;

    public static void main(String[] args) {
        ClassPathXmlApplicationContext factory = null;
        factory = new ClassPathXmlApplicationContext(
            new String[] { "applicationContext.xml" });

        NewsClient client = (NewsClient) factory.getBean("newsClient");
        client.showAllNews();

    }
    public void showAllNews() {
        Collection all = getManager().getAllNews();
        Iterator i = all.iterator();
        News n;
        while (i.hasNext()) {
            n = (News) i.next();
            System.out.println("Id : " + n.getId());
            System.out.println("Titre " + n.getTitle());
            System.out.println("Description " + n.getDescription());
            System.out.println("");
        }
    }
    public NewsManager getManager() {
        return manager;
    }
    public void setManager(NewsManager manager) {
        this.manager = manager;
    }
}

```

Bon ok, c'est super simple sauf qu'on a ici oublié un point important !

Eh ! réveillez vous, c'est un article sur la sécurité ! Alors tout cela ne va pas fonctionner car notre Webservice, il est super sécurisé, vous vous rappelez tout le boulot à faire pour cela ? Mais pas de panique, il ne reste pas grand chose à faire.

Rappelons nous d'abord que le mode d'authentification adopté est « HTTP BASIC », il va donc falloir trouver un moyen pour passer un « login / password » lors de l'appel de notre Webservice.

Pour cela, Spring propose avec la classe « `HttpInvokerProxyFactoryBean` » de définir la classe qui va se charger de l'exécution de la requête HTTP, on parle du « `httpInvokerRequestExecutor` ». Il nous faut donc une classe qui va jouer un peu le même rôle que le navigateur web dans le cas de site web et donc jouer le rôle de « client HTTP ». Et là, vive le monde open source, nous avons les bibliothèques « `jakarta-commons` » et plus particulièrement la bibliothèque `commons-httpclient` et sa classe `HttpClient`.

Il se trouve que la classe `HttpClient` ne peut pas être configurée via les mécanismes d'IoC de Spring, cela explique que nous devons créer une classe « Wrapper » et utiliser cette classe pour configurer correctement notre « `httpInvokerRequestExecutor` ».

La classe que j'ai créé est la suivante `com.devcom.acegi.http.HttpClientFactoryBean` :

```
package com.devcom.acegi.http;

import org.apache.commons.httpclient.Credentials;
import org.apache.commons.httpclient.HttpClient;
import org.apache.commons.httpclient.UsernamePasswordCredentials;
import org.springframework.beans.factory.FactoryBean;
import org.springframework.beans.factory.InitializingBean;

public class HttpClientFactoryBean implements FactoryBean, InitializingBean {

    private HttpClient httpClient;
    private String username;
    private String password;
    private String authenticationHost;
    private String authenticationRealm;

    public Object getObject() throws Exception {
        return httpClient;
    }
    public Class getObjectType() {
        return HttpClient.class;
    }
    public boolean isSingleton() {
        return true;
    }
    public void afterPropertiesSet() throws Exception {
        if ((username==null) || (password==null)) {
            throw new IllegalArgumentException("Vous devez définir le username et
le password");
        }
        httpClient = new HttpClient();
        httpClient.getState().setAuthenticationPreemptive(true);
        Credentials credentials = new UsernamePasswordCredentials(username,password);

        httpClient.getState().setCredentials(authenticationRealm,authenticationHost,credentials);
    }
    public String getAuthenticationHost() {
        return authenticationHost;
    }
    public void setAuthenticationHost(String authenticationHost) {
        this.authenticationHost = authenticationHost;
    }
    public String getAuthenticationRealm() {
        return authenticationRealm;
    }
    public void setAuthenticationRealm(String authenticationRealm) {
        this.authenticationRealm = authenticationRealm;
    }
    public HttpClient getHttpClient() {
        return httpClient;
    }
    public void setHttpClient(HttpClient httpClient) {
        this.httpClient = httpClient;
    }
    public String getPassword() {
        return password;
    }
}
```

```

    }
    public void setPassword(String password) {
        this.password = password;
    }
    public String getUsername() {
        return username;
    }
    public void setUsername(String username) {
        this.username = username;
    }
}

```

Enfin, notre « HttpInvokerProxyFactoryBean » est complété comme suit :

```

<bean id="newsManager"
      class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
    <property name="serviceUrl"
value="http://localhost:8080/newsserverws/ws/NewsManagerService.do"/>
    <property name="serviceInterface" value="com.devcom.acegi.news.NewsManager"/>
    <property name="httpInvokerRequestExecutor" ref="requestExecutor"/>
</bean>
<bean id="requestExecutor"
class="org.springframework.remoting.httpinvoker.CommonsHttpInvokerRequestExecutor">
    <property name="httpClient">
        <bean class="com.devcom.acegi.http.HttpClientFactoryBean">
            <property name="username" value="admin"/>
            <property name="password" value="admin"/>
        </bean>
    </property>
</bean>
</bean>

```

Les propriétés « username » et « password » sont ici positionnées avec le couple « admin / admin » qui permettent d'accéder au Webservice en tant qu'administrateur.

A tout moment dans l'application, il reste possible de récupérer le « requestExecutor », de changer le username et le password et de re-exécuter la méthode « afterPropertiesSet » pour créer un nouveau « HttpClient » avant de faire appel à un Webservice quelconque.

Conclusion

La sécurité, si c'est très utile, c'est pas simple, non ?

Mais bon comme il faut y passer, mieux vaut utiliser un framework sur lequel on va pouvoir capitaliser parce qu'il sera réutilisable dans divers contextes. Je ne vous ai pas montré comment sécuriser des objets autres que des services mais bon, j'imagine que vos besoins seront d'abord la sécurisation des services offerts par vos applications. Sachez aussi que pour les autorisations, ACEGI permet d'utiliser les annotations directement dans votre code Java (Annotations Java 5 ou Commons annotations), c'est probablement plus facile d'usage car vous n'avez pas à toujours modifier vos fichiers de configuration Spring.

Enfin, n'oubliez pas tous les autres filtres que j'ai mentionné en [début d'article](#), comme le « Run-As » qui vous permet de changer d'identité quand votre serveur doit appeler un autre serveur avec une identité différente de celle du client initial, et n'oubliez pas que vous pouvez aussi mettre en place un cache (ehcache par exemple) au niveau des « providers » pour éviter trop de requêtes vers votre base de données d'authentification.

Au fait, les versions des frameworks utilisés pour mes exemples sont :

- acegi-security-1.0.1
- spring-framework-2.0-rc3
- Tomcat5.5, WebSphere 6.1, JBoss 4.0.3SP1 pour les runtimes testés

Le code source utilisé pour cet article est [ici](#).

Annexe

Pour ceux qui veulent voir comment utiliser les annotations pour déclarer les autorisations, voici comment faire :

1- Dire que l'on va utiliser les annotations de type jakarta-commons (c'est ce que j'ai choisi)

```
<bean id="attributes" class="org.springframework.metadata.commons.CommonsAttributes"/>
```

2- Déclarer que les autorisations sont à rechercher via les annotations.

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

3- Modifier la définition de notre newsManagerSecurityInterceptor pour dire d'utiliser notre nouvel objet objectDefinitionSource.

```
<bean id="newsManagerSecurityInterceptor"
      class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
  <property name="authenticationManager" ref="authenticationManager"/>
  <property name="accessDecisionManager" ref="decisionManager"/>
  <property name="objectDefinitionSource" ref="objectDefinitionSource"/>
</bean>
```

4- Ajouter les annotations dans l'interface NewsManager.

```
public interface NewsManager {
    /**
     * @org.acegisecurity.SecurityConfig("ROLE_ADMIN")
     * @org.acegisecurity.SecurityConfig("ROLE_USER")
     */
    public Collection getAllNews();
    /**
     * @org.acegisecurity.SecurityConfig("ROLE_ADMIN")
     */
    public void createNews(News n);
    /**
     * @org.acegisecurity.SecurityConfig("ROLE_ADMIN")
     */
    public void deleteNews(News n);
}
```

Il faut bien entendu mettre en place le système de compilation requis pour les annotations (cf. fichier build.xml du zip contenant l'exemple).

Notez aussi que le fichier MANIFEST du jar commons-attributes-api.jar doit être patché pour y enlever la déclaration des extensions qdox et ant. Le jar fourni intègre ce patch.