SPRING – Module de sécurité Spring Security

par Samir MILOUDI

Ecoalis Consulting – samir.miloudi@gmail.com

Programme

Partie 1 : Introduction à Spring Security

Partie 2 : Gestion de l'authentification

Partie 3 : Gestion des autorisations

Partie 1

Introduction à Spring Security

- Définitions
- Présentation
- Installation
- Configuration

Définitions

- Subject
- Principal
- Ressource
- Permission

Subject

représente un utilisateur tel qu'il est vu par l'application en cours. Ce *Subject* peut posséder plusieurs *Principal*.

Principal

Login, numéro de Sécurité sociale et adresse e-mail sont des exemples de *Principal* d'un même *Subject*.

Ressource :

représente une entité protégée. Il peut s'agir d'un fichier, d'une URL ou d'un objet.

Permission

correspond au droit d'accéder à une ressource. Pour qu'un utilisateur accède à une URL protégée, il faut que ses autorisations lui donnent la permission d'y accéder.

Présentation

- API proposées par Java
- *JAAS*
- Spécification Java EE
- Spring Security

API proposées par Java

Java propose deux API pour gérer la sécurité :

1)JAAS, pour les projets Java standards (J2SE)

2)une API spécifique incluse dans la norme Java EE

JAAS

JAAS (Java Authentication and Authorization Service) est une API de bas niveau, permettant en de gérer les privilèges du code qui s'exécute.

Prévue pour les applets et les applications graphiques autonomes, elle est **peu adaptée aux applications Java EE**.

Spécification Java EE

Cette norme, réservée à la couche Web (puisque l'objet `HttpServletRequest` est requis) n'est **pas portable** entre les serveurs d'application et fournit des **services trop limités**.

Spring Security (1/2)

Spring Security, anciennement *Acegi Security*, fournit une solution déclarative, portable, plus riche et transverse aux différentes couches.

Spring Security (2/2)

C'est une extension du projet Spring pour la gestion des exigences de sécurité des applications, à savoir :

1)L'**authentification** : vérification qu'un utilisateur est bien la personne qu'il prétend être. Cette vérification s'effectue généralement à l'aide d'un couple identifiant / mot de passe.

2)La **gestion des autorisations** : vérification qu'un utilisateur authentifié a la permission de réaliser une action. Un utilisateur possède généralement un ensemble d'autorisations, qui peuvent être gérées par groupes pour plus de facilité.

Installation

- Intercepteur de sécurité

Il suffit de déclarer l'intercepteur de sécurité (filtre de *servlet*) dans le fichier `web.xml` d'une application Web :

```
<filter>
   <filter-name>springSecurityFilterChain</filter-name>
   <filter-class>
      org.springframework.web.filter.DelegatingFilterProxy
   </filter-class>
</filter>

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

l'ensemble des requêtes HTTP (pattern = /*) sera redirigé vers la classe `DelegatingFilterProxy` et filtré par un ensemble de filtres.

Configuration

- applicationContext-security.xml
- web.xml

La configuration sera de préférence effectuée dans un fichier de configuration Spring réservé à cet usage :

`WEB-INF/applicationContext-security.xml`

Le schéma XML « `security` » y sera ajouté :

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xsi:schemaLocation="
  http://www.springframework.org/schema/beans
  http://www.springframework.org/schema/beans/spring-beans.xsd
  http://www.springframework.org/schema/security
  http://www.springframework.org/schema/security/spring-security.xsd">
[...]
</beans:beans>
```

Ce fichier sera ensuite référencé dans le `web.xml` par le chargeur de définition des beans (`ContextLoaderListener`)

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext-security.xml</param-value>

  [autres fichiers de configuration (DAO, services, ...]

</context-param>

<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

Partie 2

Gestion de l’authentification

- Mise en œuvre
- Exemple avec le mécanisme le plus simple
- Exemple d'authentification à partir d'une base de données
- Exemple d'authentification à partir d'un LDAP
- Filtres

Mise en œuvre

- `AuthenticationProvider`
- `UserDetailsService`

Deux éléments sont nécessaires à la mise en œuvre de l'authentification :

1) `AuthenticationProvider` : effectue l'authentification proprement dite sur demande des autres composants du système

2) `UserDetailsService` : récupère les informations de l'utilisateur et ses rôles

AuthenticationProvider (1/2)

Cette interface peut être implémenté par :

- `LdapAuthenticationProvider` : effectue une authentification en utilisant un annuaire implémentant le protocole LDAP
- `JaasAuthenticationProvider` : effectue une authentification au moyen d'un fichier de configuration JAAS.

AuthenticationProvider (2/2)

- `CasAuthenticationProvider` : implémente une authentification fondée sur la solution de *Single Sign On* JA-SIG CAS (Central Authentication Server).
- `DaoAuthenticationProvider` : délègue la récupération des informations sur l'utilisateur à un `UserDetailsService` puis effectue la vérification du mot de passe à partir de ces informations.
- une classe personnalisée

UserDetailsService

Cette interface peut être implémentée par :

- `InMemoryDaoImpl` : récupère les informations à partir d'une structure de données en mémoire.
- `JdbcDaoImpl` : récupère les informations à partir d'une base de données.
- `LdapUserDetailsService` : récupère les informations utilisateur en interrogeant un annuaire LDAP.
- une classe personnalisée

Exemple avec le mécanisme le plus simple

Un « DaoAuthenticationProvider » délègue la récupération des informations utilisateur à un service « InMemoryDaoImpl ».

```
<bean id="daoAuthenticationProvider"
class="org.springframework.security.authentication.dao.DaoAuthenticationProvider">
  <property name="userDetailsService" ref="inMemoryDaoImpl"/>
</bean>

<bean id="inMemoryDaoImpl"
class="org.springframework.security.userdetails.memory.InMemoryDaoImpl">
  <property name="userMap">
    <value>
      user1=password,ROLE_USER
      user2=motdepasse,disabled,ROLE_ADMIN
      admin=admin123,ROLE_USER,ROLE_ADMIN
    </value>
  </property>
</bean>
```

Ce service contient une liste d'utilisateurs avec leurs mots de passe et rôle(s). Cette liste peut être externalisée dans un fichier de propriété.

Exemple d'authentification à partir d'une base de données

```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="org.hsqldb.jdbcDriver"/>
  <property name="url" value="jdbc:hsqldb:hsql://localhost:9001"/>
  <property name="username" value="sa"/>
  <property name="password" value=""/>
</bean>

<beans:bean id="myUserDetailsService"
class="org.springframework.security.userdetails.jdbc.JdbcDaoImpl">
  <beans:property name="dataSource" ref="dataSource"/>
</beans:bean>

<authentication-provider user
```

1) le bean `myUserDetailsService` est une classe de type `JdbcDaoImpl`. Il suffit de préciser la référence à la DataSource qui contient les tables de sécurité.
2) ce bean est ensuite rattaché à l'`AuthenticationProvider`

Exemple d'authentification à partir d'un LDAP

- Par Binding
- Par comparaison du mot de passe
- Utilisation d'un `AuthenticationProvider` personnalisé

Spring Security permet de s'authentifier avec un annuaire suivant deux approches :

1)par Binding : tente une connexion au LDAP avec les paramètres d'authentification de l'utilisateur. Si le *binding* réussit, l'utilisateur est considéré comme authentifié. Avec cette approche, l'authentification est finalement déléguée à l'annuaire.

2)par comparaison du mot de passe : consiste à récupérer les informations utilisateur depuis l'annuaire LDAP et à comparer le mot de passe fourni par l'utilisateur. La comparaison des informations est cette fois effectuée par l'application.

Binding (1/2)

Cette approche est réalisée en deux temps:

1)Déclaration de l'URL du serveur LDAP et de l'accès du manager :

```
<ldap-server
  url="ldap://localhost:389/"
  manager-dn="uid=admin,ou=system"
  manager-password="secret" />
```

Binding (2/2)

2)Déclaration de l'AuthenticationProvider LDAP :

```
<ldap-authentication-provider
   user-search-filter="(cn={0}) 1
   user-search-base="ou=people,o=ecoali 2
   group-search-base="ou=groups,o=ecoalis
   group-role-attribute="c 4 3
   group-search-filter="(member={0})" 5
   />
```

Un filtre de recherche (1) et un nœud de base (2) sont précisés pour la recherche récursive des utilisateurs.
Un noeud de base pour les groupes (3), l'attribut LDAP (4) qui va être utilisé par Spring Security pour définir le rôle dans l'application, et enfin le filtre de recherche permettant de retrouver l'utilisateur dans les groupes (5) sont également présents.

Spring Security passe les rôles en majuscules et leur ajoute un préfixe, qui peut être précisé par l'attribut `role-prefix` (par défaut : "ROLE_").

Comparaison du mot de passe

Il suffit d'ajouter la balise `password-compare` dans `ldap-authentication-provider` :

```
<ldap-authentication-provider
   user-search-filter="(cn={0}) »
   user-search-base="ou=people,o=ecoalis"
   group-search-base="ou=groups,o=ecoalis"
   group-role-attribute="cn"
   group-search-filter="(member={0})">

  <password-compare hash="plaintext" password-attribute="userpassword" />
</ldap-authentication-provide
```

1) l'algorithme de hashage (MD5, SHA, ...)
2) l'attribut dans lequel se trouve le mot de passe

Utilisation d'un AuthenticationProvider personnalisé

Il faut utiliser une balise spécifique dans le Bean pour que celui-ci soit intégré au mécanisme d'authentification :

```
<beans:bean id="myAuthProvider" class="custom.AuthProvider">
   <custom-authentication-provider />
   [...]
</beans:bean>
```

Filtres

- Définition
- Récupération du contexte de sécurité
- Filtre de configuration automatique
- Filtre d'interception des URL
- Filtre d'authentification avec formulaire

Filtres

- Filtre de déconnexion
- Filtre d'authentification automatique
- Filtre de connexion anonyme
- Filtre de limitation des connexions simultanées

Définition

Listés dans le balise « `http` », les différents filtres de Spring Security (*filter chain*) ont chacun une **responsabilité particulière** concernant la sécurité.

Ces filtres sont appliqués de manière **séquentielle** sur chaque URL entrante.

Récupération du contexte de sécurité

L'objet « Authentication » contient diverses informations sur l'utilisateur (identifiant, mot de passe) et ses rôles.

Par défaut, le contexte de sécurité est stocké dans une variable propre au thread courant (*thread local*).

```
Authentication auth = SecurityContextHolder.getContext().getAuthentication();
```

Filtre de configuration automatique (1/2)

La configuration suivante positionne les filtres par défaut (auto-config="true") sur toutes les URL (/**) et indique que le rôle nécessaire pour les utilisateurs authentifiés est « ROLE_USER »:

```
<http auto-config="true">
  <intercept-url pattern="/**" access="ROLE_USER" />
</http>

<authentication-provider>
   [...]
</authentication-provider>
```

Filtre de configuration automatique (2/2)

La configuration par défaut est équivalente à :

```
<http>
  <intercept-url pattern="/**" access="ROLE_USER" />
  <form-login />
  <anonymous />
  <http-basic />
  <logout />
  <remember-me />
</http>

<authentication-provider>
   [...]
</authentication-provider>
```

Tous ces filtres sont ajoutés implicitement lors d'une configuration « auto »

Filtre d'interception des URL

La balise « `intercept-url` » permet de définir les règles d'interception à appliquer sur un ensemble d'URL, selon une notation proche de celle utilisée par *Ant* pour la désignation des fichiers.

Exemples :

```
/** : l'ensemble des URL
/admin/** : l'ensemble des URL se trouvant sous
            le chemin « admin » ainsi que ses sous-chemins
/*delete* : toutes les URL contenant le mot "delete".
```

L'attribut « `requires-channel` » permet d'exiger le mode sécurisé (https)

Filtre d'authentification avec formulaire (1/2)

Toutes les URL protégées (balise « `intercept-url` ») seront automatiquement redirigées vers un formulaire d'authentification si l'utilisateur n'est pas signé lorsqu'il effectue une action.

Filtre d'authentification avec formulaire (2/2)

```
<http>
  <form-login
        login-page="/login.htm"
        authentication-failure-url="/authenticationFailure.htm"
        default-target-url="/welcome.htm"
        login-processing-url="/j_my_authentication_check"
  />
  [...]
</http>
```

1) `login-page` : URL affichant le formulaire de connexion.
   Ce formulaire doit contenir les champs `j_username` et `j_password` et être soumis vers « `login-processing-url` ».
2) `authentication-failure-url` : URL vers laquelle Spring Security redirige en cas d'échec de l'authentification.
3) `default-target-url` : URL vers laquelle l'utilisateur est redirigé si l'authentification a réussi.
4) `login-processing-url` : URL sur laquelle Spring Security attend les paramètres d'authentification.
   Par défaut, cette URL est `j_spring_security_check`.

Filtre de Déconnexion

Le contexte de sécurité de l'utilisateur sera vidé lorsque cette URL sera appelée.

```
<http>
  <logout
    logout-success-url="/logoutSuccessful.jsp"
    logout-url="/j_my_logout" />
  [...]
</http>
```

1) `logout-success-url` : URL vers laquelle l'utilisateur est redirigé après la déconnexion.
2) `logout-url` : URL de déconnexion (par défaut `j_spring_security_logout`).

Filtre d'authentification automatique

Pour proposer la connexion automatique, Spring envoie un cookie crypté au navigateur de l'utilisateur.

Un autre mécanisme d'authentification automatique proposé est fondé sur la persistance en base de données du *jeton d'authentification*.
Ce système est globalement plus fiable car le cookie change à chaque connexion. Il faut alors préciser la *DataSource* :

```
<http>
  <remember-me data-source-ref="dataSource" />
  [...]
</http>
```

Filtre de connexion anonyme

Ce filtre permet de positionner un contexte de sécurité par défaut, avec un nom d'utilisateur et un ou plusieurs rôles.

Le contexte de sécurité ne sera donc jamais nul.

```
<http>
   <anonymous username="guest" granted-authority="ROLE_INVITE" />
   [...]
</http>
```

Filtre de limitation des connexions simultanées

Ce filtre nécessite l'ajout d'un *listener* dans `web.xml` :

```
<listener>
  <listener-class>org.springframework.security.ui.session.HttpSessionEventPublisher</listener-
class>
</listener>
```

```
<http>
  <concurrent-session-control
   max-sessions="1" exception-if-maximum-exceeded="true" />
  [...]
</http>
```

Si l'attribut "exception-if-maximum-exceeded" est à "false" : lorsque le nombre maximum de sessions pour un même utilisateur est atteint, elle sont toutes invalidées, sinon l'utilisateur ne peut simplement pas se connecter.

Partie 3

Gestion des autorisations

- Sécurisation des vues
- Sécurisation de l'invocation des méthodes

Sécurisation des vues

- Introduction
- Configuration
- Interface `Authentication`
- Balise `security`

Introduction

Spring propose un ensemble de balises JSP qui permettent d'accéder au *contexte de sécurité*.

Elle permettront de sécuriser certains éléments des pages (liens, boutons d'action, ...).

Cette démarche complètera la sécurisation par interception d'URL, qui fonctionne en tout ou rien.

Configuration

La bibliothèque de balises JSP de Spring Security devra être inclue dans toute page JSP y recourant :

```
<%@ taglib prefix="security" uri="http://www.springframework.org/security/tags" %>
```

Interface Authentication

L'accès à l'interface `Authentication`, permettant de récupérer les informations sur l'utilisateur et ses rôles, est réalisé avec la balise « `security:authentication` » :

```
Bienvenue <security:authentication property="principal.username" /> !

<security:authentication property="authorities" var="authorities" />
```

Récupère le nom de l'utilisateur connecté

Récupère l'ensemble des rôles de l'utilisateur connecté dans une variable

Balise security

La balise « security:authorize » ne génèrera son contenu en HTML que si la condition de rôle est remplie :

```
<security:authorize ifAllGranted="ROLE_ADMIN,ROLE_USER">
    à la fois admin ET utilisateur.
</security:authorize>

<security:authorize ifAnyGranted="ROLE_ADMIN,ROLE_USER">
    pour les admins OU les utilisateurs.
</security:authorize>

<security:authorize ifNotGranted="ROLE_ADMIN,ROLE_USER">
    ni pour les admins, ni pour les utilisateurs.
</security:authorize>
```

Sécurisation de l'invocation des méthodes

- Introduction
- Sécuriser avec des annotations
- Sécuriser un Bean
- Sécuriser un ensemble de Beans

Introduction

Spring Security propose trois approches pour sécuriser l'invocation des méthodes :

1)en apposant des annotations directement sur les classes ou les méthodes à protéger

2)lors de la déclaration d'un Bean

3)en définissant des coupes avec la syntaxe AspectJ

Sécuriser avec des annotations (1/2)

Les annotations utilisées sont soit celles de la JSR 250, soit celles de Spring : les `secured-annotations`.

Il faut tout d'abord déclarer leur utilisation dans le fichier de contexte :

```
<global-method-security jsr250-annotations="enabled" />
            ou
<global-method-security secured-annotations="enabled" />
```

Sécuriser avec des annotations (2/2)

Les méthodes sont ensuite annotées avec l'une ou l'autre des normes

```
public class UserManagerImpl implements UserManager {

   @RolesAllowed("ROLE_ADMIN")
   ou
   @Secured("ROLE_ADMIN")
   public void createUser(User user)
         throws UserAlreadyExistsException {
      [...]
   }
   @RolesAllowed({"ROLE_ADMIN","ROLE_USER"})
   ou
   @Secured({"ROLE_ADMIN","ROLE_USER"})
   public User findUser(String login) {
      [...]
   }
   [...]
```

Tout accès à une méthode sécurisée sera intercepté (Beans gérés par Spring!) et le contexte de sécurité sera utilisé pour déterminer si les rôles de l'utilisateur lui permettent d'exécuter l'action.

Sécuriser un Bean

Un Bean peut être sécurisé en même temps que sa déclaration :

```
<beans:bean id="userManager" class="com.ecoalis.service.impl.UserManagerImpl">
  <intercept-methods>
     <protect method="createUser" access="ROLE_ADMIN" />
     <protect method="findUser" access="ROLE_ADMIN,ROLE_USER" />
  </intercept-methods>
</beans:bean>
```

Sécuriser un ensemble de Beans

Les sécurisations à apporter seront définies dans des balises de coupe « protect-pointcut ». Ces balises précisent le ou les rôles nécessaires pour la coupe :

Cette manière de définir la sécurité des appels aux méthodes est la plus puissante et n'est pas intrusive.

```
<global-method-security>
  <protect-pointcut
   expression="execution(* com.ecoalis.service.impl.UserManagerImpl.createUser(..))"
   access="ROLE_ADMIN" />

  <protect-pointcut
   expression="execution(*com.ecoalis.service.impl.UserManagerImpl.findUser(..))"
   access="ROLE_ADMIN,ROLE_USER" />
</global-method-security>
```