DOSSIER DE SECURISATION DES WEB SERVICES

ANNEXE 5 : ETUDE DE SPRING SECURITY

ABREVIATIONS

Abréviation	Signification
SOAP	Simple Object Access Protocol
HTTP	Hyper Text Transfer Protocol
AOP	Programmation Orientée Aspect
EJB	Enterprise JavaBeans
URL	Uniform Resource Locator
JDK	Java Development Kit
JSP	JavaServer Pages
JVM	Java Virtual Machine
J2EE	Java Enterprise Edition
JAR	Java ARchive
DTD	Document Type Definition
XML	eXtensible Markup Language
LDAP	Lightweight Directory Access Protocol
POJO	Plain Old Java Object

REFERENCES

Abréviation	Signification
[DOSSIERSECWS]	Dossier_Securisation_Web_Services_v1r0.pdf
[ANASEC]	Annexe1_Analyse_Sécurité_Préalable_v1r0. pdf
[FILTRAGE]	Annexe2_Etude_ Securistation_WS_FiltrageIP_v1r0. pdf
[SSL]	Annexe3_Etude_ Securistation_WS_TunnelSSL_v1r0. pdf
[SYNAPSE]	Annexe4_Etude_ Securistation_WS_Synapse_v1r0. pdf
[SPRING]	Annexe5_Etude_ Securistation_WS_SPRING-Security_v1r0. pdf
[PROTOTYPE]	Annexe6_Etude_ Securistation_WS_Prototype_v1r0.pdf

TABLE DES MATIERES

TABLE DES ILLUSTRATIONS

TABLE DES LISTINGS

1.AVANT PROPOS

Ce document constitue une annexe du « Dossier de sécurisation des Web Services » publié par l'AMUE. Il a pour objectif de présenter la mise en œuvre de SPRING-Security dans un objectif de sécurisation des Web Services. Par conséquent, afin d'appréhender correctement le contexte et le contenu de ce document, il est conseillé de lire au préalable le dossier de sécurisation des Web Services [DOSSIERSECWS].

Cette annexe a la particularité de formuler une spécification détaillée de mise en œuvre de cette technologie.

2.DESCRIPTION GENERALE DE LA TECHNOLOGIE

ACEGI est un framework permettant d'introduire une couche de sécurité pour les Web Services en ce qui concerne le framework SPRING. ACEGI Security est devenu SPRING Security, projet officiel de sécurité de SPRING. Il est donc conseillé pour la mise en place de nouveaux environnements, de se baser sur SPRING Security. Il fournit un niveau de sécurité pour les applications J2EE. Le principe mis en place par SPRING Security pour sécuriser une ressource est relativement simple, une série de « filtres » est interposée entre l'appelant et la ressource elle-même. Ces différents filtres ont chacun un rôle précis dans la chaîne de sécurisation.

SPRING Security arbore de nouvelles fonctionnalités vis-à-vis d'ACEGI dont principalement :

- La simplification de la configuration ;

- L'intégration du système d'authentification unique (SSO) ;

- Le support du langage d'expression d'AspectJ ;

- L'amélioration du module de sécurité pour les instances d'objets basés sur les ACL (Access Control List) ;

- L'ajout de hiérarchie de rôles (équivalente aux groupes) ;

- L'amélioration du support de WSS (WS-Security).

SPRING Security permet essentiellement de gérer deux choses :

- L'authentification, qui consiste à garantir que l'entité connectée est bien celle qu'elle prétend être ;
- Les autorisations, qui consistent à vérifier que l'entité connectée a bien les permissions d'effectuer une action donnée.

Le concept central de base utilise la notion de **filtres** autour des servlets pour l'authentification et d'un mécanisme d'interception pour gérer les autorisations.

2.1. L'AUTHENTIFICATION

La première étape de la sécurisation d'une application web est l'authentification des appelants qui la manipulent. Contrairement à un site web, généralement ouvert à tous, les applications web nécessitent souvent que l'identité de l'appelant soit connue. Ceci permet entre autres, la personnalisation des services qui lui sont offerts. Pour cela, il est nécessaire de mettre en place un mécanisme de login/password permettant de manipuler les informations propres à l'appelant, ses droits, éventuellement ses groupes, etc. Cette authentification se fait à partir d'informations stockées dans une base de données, un fichier XML, un annuaire LDAP, etc.

2.2. LES AUTORISATIONS

Les éléments d'une application pouvant être sécurisés sont :

- Les URLs, c'est à dire les pages et les servlets ;
- Les méthodes des 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 (Développement spécifique).

2.2.1. SECURISATION DES ACCES A 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 appelants d'une certaine catégorie. Par exemple, dans le cadre d'une application nécessitant une authentification, l'accès des pages de l'application aux appelants non connectés doit être interdit. Il peut également être nécessaire de restreindre l'accès à certaines URLs à un groupe très particulier d'appelants connectés disposant de certains droits. C'est le cas de la partie administration d'une application web par exemple.

2.2.2. SECURISATION DES APPELS A UNE METHODE

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 appelants 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.

Spring Security propose cette sécurisation par un ensemble de mécanismes non intrusifs basés sur des Dynamic Proxy java et les concepts d'AOP (Programmation Orientée Aspect) grâce à Spring AOP. Nous verrons par la suite une implémentation simple d'une sécurisation de ce type.

2.2.3. SECURISATION DE LA MANIPULATION DES OBJETS DU MODELE

C'est le dernier point de sécurité que nous aborderons ici. Il s'agit de l'un des plus complexes à mettre en œuvre et de loin le plus coûteux en terme de performances. Il correspond à la nécessité de répondre à des exigences de sécurité très fortes et nécessite un investissement plus important. Tout comme nous avons explicité la sécurité des appels de méthodes, il s'agit ici de sécuriser les instances d'objets du modèle. Ce niveau de sécurité est rarement utilisé et ne sera pas exposé dans ce document.

2.3. PRE REQUIS TECHNIQUES

Le premier pré-requis consiste à disposer du framework Spring. Dans ce guide, nous utiliserons la version 2.0.8 de SPRING. La version de Spring Security utilisé sera la 2.0.3.

Ces modules sont téléchargeables aux URL suivantes :

- Spring: http://www.springframework.org/download/ . Après avoir dézippé le fichier téléchargé, récupérer les modules *spring.jar* et *spring-aspects.jar* sis dans le répertoire spring-framework-2.0.8/dist.
- Spring Security : http://www.springframework.org/download/ . Après avoir dézippé le fichier téléchargé, récupérer le module *spring-security-core-2.0.3.jar* sis dans le répertoire spring-security/dist.

3.LA SECURITE AVEC SPRING SECURITY : USERNAME + PASSWORD

3.1. CAS PRATIQUE : WEB SERVICES APOGEE

L'étude de la sécurité a été réalisée conjointement à un prototype intégrant les aspects de sécurisation étudiés notamment ceux de Spring Security. La couche de sécurité se veut surtout être transverse au code existant et ne doit en aucun cas interférer ou nécessiter une modification des classes existantes, donc aucun redéveloppement. Fort de ces considérations, le développement a été axé sur un module totalement transverse, s'appuyant néanmoins sur les fichiers de configuration des Web Services Apogée.

3.1.1. UNE ORGANISATION EN COUCHES

La figure suivante expose l'architecture des Web Services Apogée. Nous y remarquons une organisation en couches :

- Le consommateur. Il s'agit du client, il contient les Stubs des Web Services. Il s'agit des différentes classes générées du côté Client qui lui permettent de pouvoir consommer les Web Services distants.
- Le War Web Services. Cette couche concerne le côté serveur où on note divers modules dont la couche Web Services qui dialogue avec le consommateur. Ensuite, la couche service permettant de fournir un service particulier. Enfin, la couche Domaine et la couche Persistance qui comme l'indique son nom, a une action éponyme, c'est-à-dire s'occuper de persister les données en base de données.
- La base de données Apogée.
- Le module Spring Security viendra se greffer sur cet ensemble.

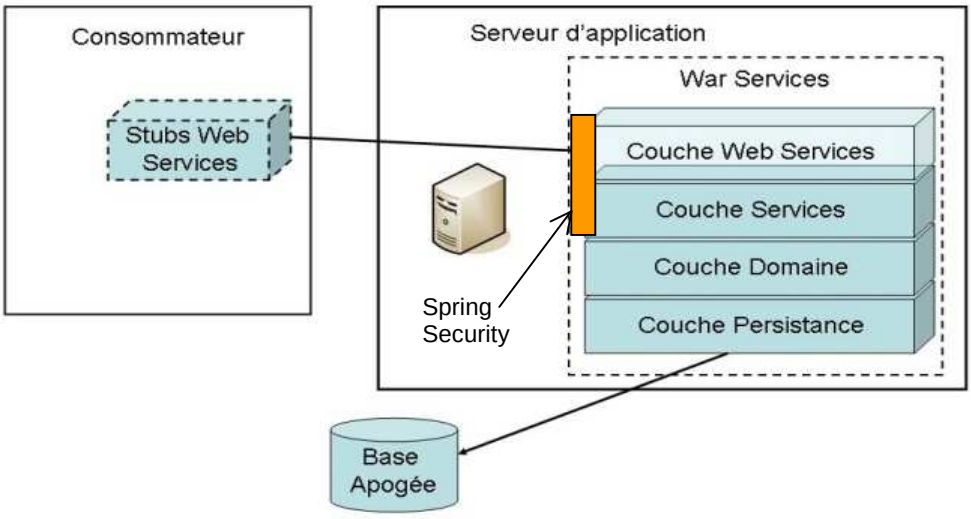

Figure 1 – L'architecture des web services Apogée.

Notre étude se focalisera particulièrement sur les couches Consommateur, Web Services et Services.

3.1.2. CONTRAINTES TECHNIQUES

Les Web Services Apogée ont été conçus en respectant un certain nombre de contraintes parmi lesquelles :

- Un Java J2SE Development Kit (JDK) version 1.4.2 ;
- Un serveur d'application respectant les spécifications Servlet 2.4 et JSP 2.0 (norme J2EE 1.4);
- Un moteur Axis 1.4 pour les Web Services.

Afin de pouvoir s'insérer sur l'existant des Web Services Apogée ces contraintes ont été respectées lors du développement du projet transverse de sécurité.. Nous nous sommes donc portés sur l'utilisation d'un JDK 1.4 et d'un serveur Tomcat 5.0 avec une JVM 1.4.

3.1.3. PERIMETRE

Afin de faciliter l'analyse de la solution SPRING Security et la constitution d'un prototype intégrant la sécurité des Web Services Apogée, il a été décidé de mener ces travaux sur un service en particulier : le service Géographie et une de ses opérations : La récupération de la liste des pays, *recupererPays()*. Côté service, une méthode bouchonnée permet alors de retourner un pays au consommateur. Pour des raisons de facilité, la méthode est dite bouchonnée car le prototype n'interroge pas directement une base Apogée. L'accès à cette base a été bloqué et une réponse statique a été intégrée dans le code. De plus, la solution proposée a été prévue pour être « débrayable » (activer ou non la sécurité).

3.2. REFERENCES

Ce document se focalise principalement sur les notions de sécurité à apporter à l'existant. Il n'abordera pas évidemment les détails des composants serveurs et logiciels, les principales fonctionnalités de ces Web Services. Nous vous invitons alors à vous référer aux différents documents portant sur ces sujets, à savoir :

- Le Manuel Technique ;
- Le Contrat du Service Géographie.

3.3. PARAMETRAGES

La mise en place des notions de sécurité passe par un certain nombre d'étapes définies dans les sections suivantes de ce document. Chaque section se veut être un enchaînement des précédentes, formant un tout. Dans chacune des étapes, il sera fait un état des différents paramétrages à effectuer, un extrait de code source éventuel viendra en éclaircir la mise en œuvre.

Coté Serveur :

- L'étape 1, concernera les préliminaires à savoir les librairies, la logique métier Apogée, et les différentes classes transverses devant permettre la sécurité ;
- L'étape 2, concernera la mise en œuvre des providers d'authentification en expliquant le Basic Handler personnalisé ;
- L'étape 3, concernera la mise en œuvre des intercepteurs pour les autorisations avec l'utilisation des notions d'AOP;
- L'étape 4, concernera le descripteur de déploiement ;
- L'étape 5, quant à elle, exposera les réadaptations à appliquer à l'existant.

Coté Client : l'étape 6 concernera le paramétrage des appels authentifiés et autorisés.

3.3.1. ÉTAPE 1 (SERVEUR) : LES PRELIMINAIRES

Cette étape permet de définir les différents éléments dont nous avons besoin pour mener à bien cette implémentation. Il s'agira des différentes librairies à installer, d'un rappel sommaire de la logique métier et d'un exposé des classes de sécurité. Cette mise en oeuvre impactera quelques couches et certains fichiers de configuration de l'existant. A ce propos, en voici les grandes lignes :

- Modification du descripteur de déploiement web.xml pour y ajouter les références vers les fichiers de configuration Spring à utiliser. Ajout des Listeners (écouteurs), qui nous permettront de récupérer le contexte de l'application quelle que soit la couche ;
- Ajout des classes transverses de sécurisation à l'arborescence du projet Apogée : Le package *springsecurity* ;
- Modification du fichier Spring concernant le périmètre de mise en œuvre, notamment le service géographie nommé *MyGeographie-SpringContext.xml* et sis dans le répertoire *src* du projet. On y définira les règles de sécurité.
- Modification du fichier *server-config.wsdd* (Web Service Deployment Descriptor) fichier de description du déploiement des Web Services. Ce fichier indique à Axis quel service doit être exposé avec les informations pour accéder à la classe du service positionnée en tant que sous-élément(s) paramètre(s) (className, allowedMethods,...). On y fera la jonction nécessaire avec un Handler défini dans le package *springsecurity*.

- Modification du code du consommateur pour intégrer le nom de l'appelant et le mot de passe dans la requête SOAP.

3.3.1.1. Les librairies

De nouvelles librairies ont été rajoutées aux existantes dans le répertoire /WEB-INF/lib. Il s'agit des JARs suivants :

- spring-security-core-2.0.3.jar , version 2.0.3 : Framework Spring Security ;
- spring.jar, version 2.0.8 : Framework Spring ;
- spring-aspects.jar, version 2.0.8 : Framework Spring gérant les notions d'aspect ;
- aspectweaver.jar (org/aspectj/weaver), version 1.5.3 : Framework contenant le mécanisme d'intégration appelé le « *weaver* ». Le module « *weaver* », à l'image du métier à tisser, compose le système final sur base de règles et les modules qui lui sont donnés ;
- commons-codec-1.3.jar, version 1.3 : Cette librairie contient les utilitaires d'encodage ;
- jdom.jar, version 1.1 : Cette librairie propose une interface qui définit la façon d'utiliser XPath en Java pour parcourir les fichiers XML. Cette notion n'étant pas innée à la version du JDK 1.4, nous avions eu recours à ce JAR.

3.3.1.2. Rappels sur la logique métier actuelle

Le Web Service de base est dans notre cas l'interface *GeographieWebServiceInterface* dans le package *gouv.education.apogee.commun.webservices*. Son implémentation : *GeographieWebServiceImpl* se trouve au niveau inférieur de ce même package, c'est-à-dire : *gouv.education.apogee.commun.webservices.impl*. Ce dernier fait appel à des services pour effectuer l'action escomptée.

Ainsi, la classe *GeographieWebServiceImpl* récupère le bean nommé « *geographieMetierService* » dans le fichier de configuration Spring « *communServicesMetier-SpringContext.xml* ». Ce bean utilise un intercepteur pour la gestion des transactions au niveau des services métiers. Il pointe sur un target : « *geographieMetierServiceTarget* » qui n'est rien d'autre que la classe d'implémentation du service métier *GeographieMetierServiceImpl* du package *gouv.education.apogee.commun.servicesmetiers.impl*. Ce target instancie par la même occasion le bean « *referentielGeographieService* » défini dans le fichier de configuration Spring « *geographie-SpringContext.xml* ». Il s'agit là du bean référençant la classe *ReferentielGeographieServiceImpl* du package *gouv.education.apogee.commun.geographie.services.impl*. Comme « *geographieMetierServiceTarget* », ce bean utilise aussi un intercepteur pour la gestion des transactions. Il instancie à son tour d'autres beans à savoir :

- *paysService*, donnant la classe d'implémentation *PaysServiceImpl* avec un cadre transactionnel. Il fournit tous les services portant sur les pays ;
- *departementService* donnant la classe d'implémentation *DepartementServiceImpl* avec un cadre transactionnel. Il fournit tous les services portant sur les départements ;
- *communeService* donnant la classe d'implémentation *CommuneServiceImpl* avec un cadre transactionnel. Il fournit tous les services portant sur les communes.

La figure suivante expose la collaboration entre les différents beans en présence.

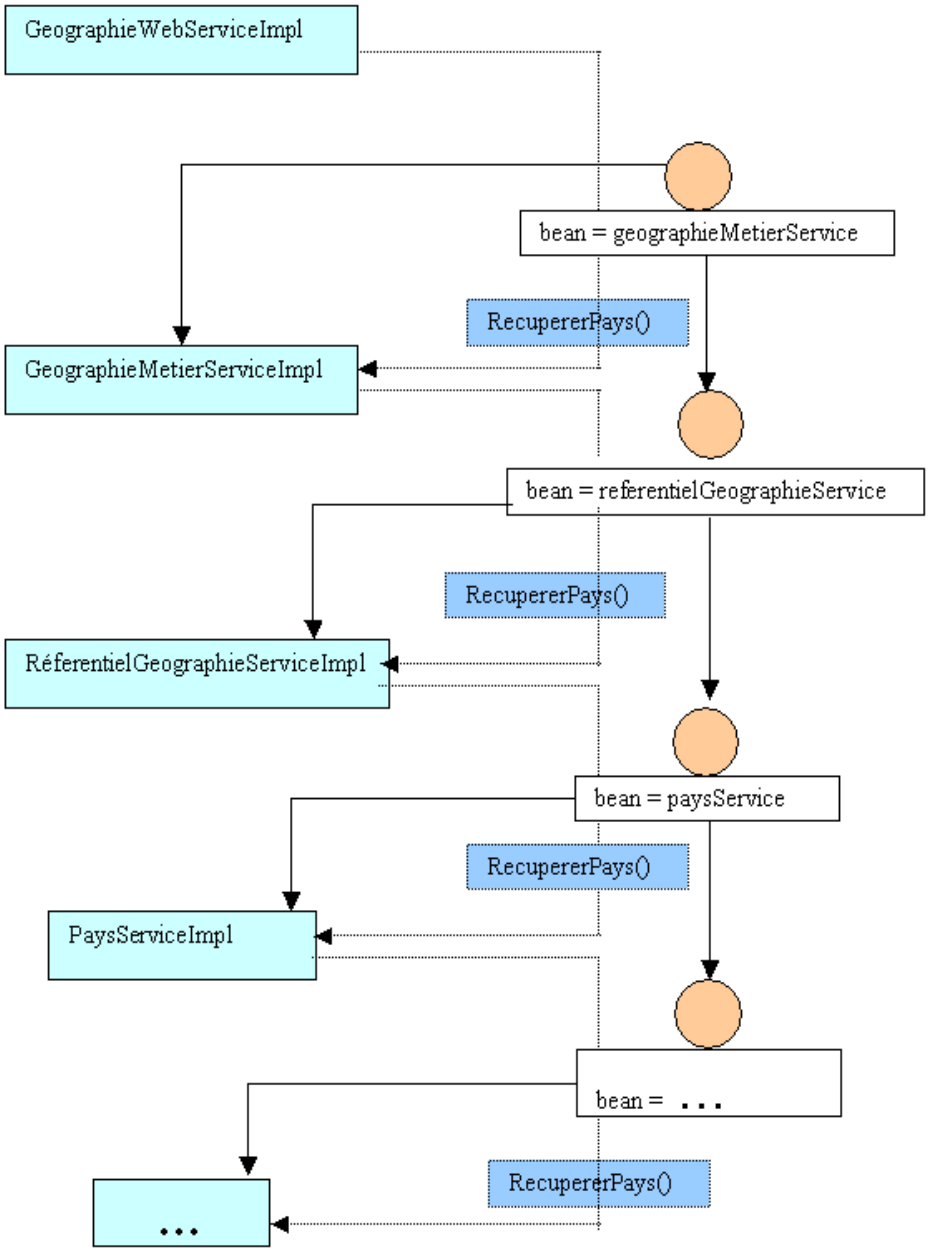

Figure 2 – La collaboration entre les différents beans dans Apogée.

3.3.1.3. Paramétrages, le web.xml : Le métronome

Il s'agit du descripteur de déploiement, se trouvant dans le répertoire ***WEB-INF***. Ce fichier contient des informations de configuration pour l'application. C'est un fichier XML avec une DTD normalisée. Il contrôle l'enregistrement des servlets, la correspondance des URL, les fichiers d'accueil, ainsi que les formalités avancées comme les contraintes de sécurité.

Dans notre cas, le fichier ***web.xml*** existant a dû subir une légère modification pour permettre le branchement des classes transverses de sécurisation.

Les principales modifications sont en bleu dans le listing de code suivant. Dans le premier, compte tenu du périmètre sur les web services Apogée, se limitant qu'au web service Géographie, nous n'avons alors chargé que le fichier de configuration Spring associé. Nous l'avons nommé *MyGeographie-SpringConfig.xml* (expliqué plus loin dans ce document) et *persistance-SpringContext.xml* qui est l'apanage des connexions à la base de données et qui fournit un bean Spring particulier : le *transactionManager*, gestionnaire transactionnel Spring.

Dans le listing suivant, concernant toujours le fichier web.xml, nous avons défini un listener personnalisé, ***StartupListener***, dont le rôle est de nous fournir une instance du contexte de l'application une fois chargée. Pour fonctionner, ce listener nécessite la mise en place du listener de base de Spring, le ***ContextLoaderListener***. Notons ici que l'ordre de définition de ces listeners doit être respecté.

```
<context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>
                <!-- On spécifie ici les fichiers de configuration Spring à charger-->
                classpath:/MyGeographie-SpringContext.xml
                classpath:/persistance-SpringContext.xml
        </param-value>
</context-param>
…
<listener>
        <listener-class>
                org.springframework.web.util.Log4jConfigListener
        </listener-class>
</listener>
<servlet>
        <servlet-name>SpringContextServlet</servlet-name>
        <servlet-class>
                org.springframework.web.context.ContextLoaderServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>
<!--Définition du listener de base du framework de Spring-->
<listener>
        <listener-class>
                org.springframework.web.context.ContextLoaderListener
        </listener-class>
</listener>
<!--Définition d'un listener customisé pour récupérer le contexte spring chargé-->
<listener>
        <listener-class>
                gouv.education.apogee.commun.securite.springsecurity.utils.StartupListener
        </listener-class>
</listener>
```

<u>Listing 1</u> – Un extrait du fichier web.xml.

3.3.1.4. Quid les classes transverses de la sécurité

Les différentes classes nécessaires au mécanisme de sécurisation, devront se greffer facilement sur le projet existant. Pour ce faire, un package sera créé dans l'arborescence du projet (Projet Eclipse en l'occurrence). Nous avons fait le choix de nommer ce package *springsecurity* au chemin suivant : *gouv.education.apogee.commun.securite*. Il comporte les classes suivantes :

- ***SpringSecurityBridgeAuthenticationHandler.java***;

- ***User .java*** ;
- ***UserManager.java*** ;
- ***UserManagerImpl.java*** ;
- ***SpringContext.java*** (package utils) ;
- ***StartupListener.java*** (package utils) ;
- ***XmlUtils.java*** (package utils) ;

L'arborescence est visible sur la figure suivante.

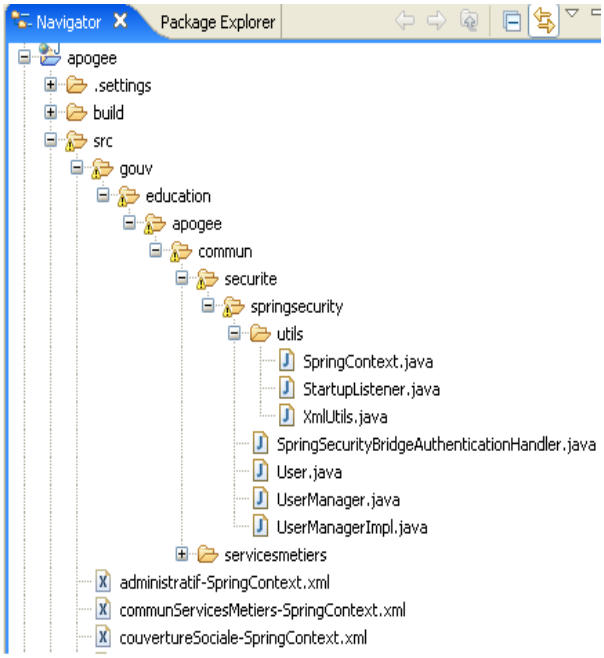

Figure 3 - L'arborescence des classes transverses.

3.3.1.4.1.Les Utilitaires

a. ***SpringSecurityBridgeAuthenticationHandler.java*** : Il s'agit d'un Handler customisé responsable de la sécurité du Web Service. Il étend la classe Axis *BasicHandler*. Comme l'indique son nom, il agit comme un pont entre le moteur Axis et Spring Security. C'est une classe personnalisée devant être le garant de la sécurité. A partir de la récupération du bean « *userManager* » (bean spring fournissant une implémentation à l'interface *UserManager*), il permettra de « logger » l'entité et de créer un jeton particulier gage de l'authentification.

b. ***User.java*** : Il s'agit de la classe représentant un appelant au sens entité désirant consommer le Web Service. Il dispose des attributs suivants : username, qui représente le nom d'appelant, password, représentant le mot de passe et enfin de « *credentials* », qui est un tableau contenant les différents rôles de l'entité.

c. ***UserManager.java*** et ***UserManagerImpl.java*** : elles désignent respectivement l'interface et la classe d'implémentation du manager ayant pour rôle de gérer un User (entité). Cette gestion n'est

rien d'autre que le « login » de ce dernier. La méthode login permet, à partir des paramètres username et password qui lui sont passés en paramètre, d'en vérifier l'existence via un fichier XML contenant la liste des appelants et leurs rôles. Dans le cas où l'entité y est déclarée, ses rôles sont alors récupérés. Un objet *User* décrivant cette entité est alors réacheminé vers le programme appelant.

Dans le sous package *utils*, on distingue :

a. ***StatupListener.java*** : classe utilitaire de gestion du contexte Spring. Il s'agit de la classe du listener customisé défini dans le fichier web.xml relaté dans les sections antérieures. Comme en indique le nom, cette classe implémentant l'interface ***ServletContextListener***, définit la méthode ***contextInitialized*** (initialisation du contexte Spring) et agit comme un écouteur. Lors de l'initialisation du contexte Spring, la méthode ***contextInitilized*** est appelée et une instance du contexte est alors fournie à la classe ***SpringContext.java***.

b. ***SpringContext.java*** : classe utilitaire de mise à disposition du contexte Spring récupéré. Il s'agit d'un singleton. Une instance du contexte Spring lors de son initialisation lui est fournie. Ceci permet aux classes devant utiliser cette référence au contexte, de la récupérer à partir de la classe ***SpringContext.java***. Il est important de signaler ici le but d'une telle manœuvre : La récupération du bean « *userManager* » dans la classe ***SpringSecurityBridgeAuthenticationHandler*** donnant une implémentation de l'interface *UserManager*. Ce mécanisme de récupération du contexte Spring en facilite largement la mise en œuvre.

c. ***XmlUtils.java*** : classe utilitaire de gestion de fichiers XML. Elle permet de récupérer un document Dom à partir d'un fichier XML en entrée. Elle dispose aussi d'une méthode permettant d'encoder un mot de passe.

3.3.1.4.2.Le fichier des appelants

La vérification de la chaîne de connexion d'un consommateur peut s 'effectuer à l'aide d'une base de données, d'un serveur LDAP ou même d'un fichier à plat. Dans le cas de notre étude, cette vérification se fera avec cette dernière option. Il s'agit d'un fichier XML contenant les informations sur les différentes entités habilitées à consommer les Web Services Apogée. Il doit être inséré à la racine du package ***src***, au même niveau que les fichiers de configuration de Spring. Il est recopié lors de la compilation dans le répertoire ***WEB-INF/classes***. Il contient les informations telles que le username, le password et les rôles de l'entité. Notons que le password est encrypté avec SHA. Le listing suivant représente le fichier des entités que nous nommerons ***users.xml***.

```
<users>
        <!-- Définition d'un user-->
        <user>
                <!-- Définition de son nom d'utilisateur-->
                <username>myUser</username>
                <!-- Définition de son mot de passe encrypté par SHA-->
                <password> 639e810369f37474215d1af2ca701baa76d6ca7c</password>
                <!-- Définition des roles-->
                <roles>
                        <!-- Définition d'un rôle administrateur : ROLE_ADMIN-->
                        <role>ROLE_ADMIN</role>
                        <role>ROLE_MANAGER</role>
                </roles>
        </user>
        <user>
                <username>myUser1</username>
                <password> dd6e21c739ac6abb8200f8c583bd1baa0b96b282</password>
                <roles>
                        <role>ROLE_ADMIN</role>
                        <role>ROLE_MANAGER</role>
                </roles>
        </user>
</users>
```

Listing 2 – Un extrait du fichier des entités déclarées : users.xml.

3.3.1.4.3.Intégration

Toutes les classes ci-dessus listées sont intégrées transversalement à l'application dans un package dénommé ***springsecurity***. L'intégration dans l'arborescence se fait en collant ce répertoire dans le package *securite* des Web Services Apogée. Puisque toutes les sources du projet sont compilées et réorganisées sous forme de fichiers JAR, il est alors nécessaire de créer le chemin complet dans un projet Eclipse par exemple. Ainsi, dans le répertoire ***src*** de l'application, créer successivement les packages suivants : *gouv.education.apogee.commun.securite*. Le package ***springsecurity*** sera donc inséré dans le chemin ainsi créé. Le fichier XML *users.xml* quant à lui pourra être déposé dans le répertoire WEB-INF de l'application.

Les différentes interactions entre ces différentes classes sont exposées dans la figure suivante :

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

web.xml

```
public class StartupListener implements ServletContextListener {
  /** context */
  private ServletContext context;
  public void contextInitialized(ServletContextEvent sce) {
      context = sce.getServletContext();
      XmlWebApplicationContext ctx = null;
      try {
          ctx = (XmlWebApplicationContext) context
                  .getAttribute(WebApplicationContext.
              ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE);
      } catch (Exception e) {
          e.printStackTrace();
      }
      SpringContext.getInstance().setCtx(ctx);
  }
```

StartupListener.java

```
package gouv.education.apogee.commun.securite.springsecurity.utils;
public class SpringContext {
    public static SpringContext getInstance() {

        if (springContext == null) {
            springContext = new SpringContext();
        }
        return springContext;
    }
   /**
    * @return the ctx
    */
   public XmlWebApplicationContext getCtx() {
       return ctx;
   }
   /**
    * @param ctx
    *            the ctx to set
    */
   public void setCtx(XmlWebApplicationContext ctx) {
       this.ctx = ctx;
   }
```

SpringContext.java

```xml
<bean id="userManager"
    class="gouv.education.apogee.commun.securite.springsecurity.UserManagerImpl">
</bean>
```

MyGeographie-SpringContext.xml

```
public class SpringSecurityBridgeAuthenticationHandler extends BasicHandler {
 public void invoke(MessageContext msgContext) throws AxisFault {
     UserManager userManager = (UserManager) SpringContext.getInstance()
             .getCtx().getBean("userManager");

     User user = userManager.login(msgContext.getUsername(), msgContext
             .getPassword());
   ...
 }
```

SpringSecurityBridgeAuthenticationHandler.java

Figure 4 - Les interactions entre les classes transverses de sécurité.

3.3.1.5. Les modules en présence

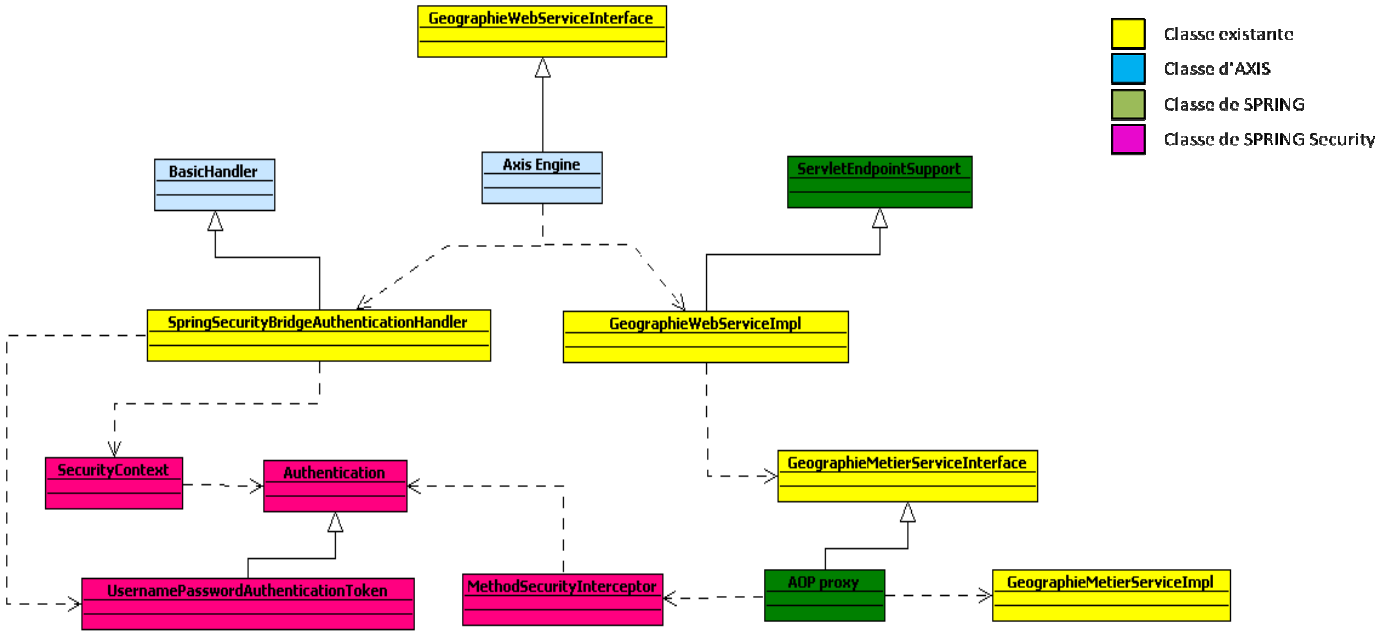

Figure 5 – Le diagramme des classes des modules en présence.

La figure précédente expose la mise en œuvre. Les objets en jaune sont ceux que nous implémenterons. *GeographieWebServiceInterface*, *GeographieWebServiceImpl*, *GeographieMetierServiceInterface* et *GeographieMetierServiceImpl* sont des classes existantes. Néanmoins, la méthode *recupererPays()* de la classe *GeographieMetierServiceImpl* a été bouchonnée afin d'éviter l'accès à la base de données. Les objets en bleu sont ceux d'Axis. Les roses, Spring Security, et les verts ceux de Spring. *GeographieWebServiceInterface* est l'interface de notre Web Service. Le diagramme est simplifié en intégrant tous les composants Axis dans le composant Axis Engine. Le *BasicHandler* est aussi une classe Axis, mais il est exposé séparément. *GeographieWebServiceImpl* est une classe pouvant être générée par Axis mais dont le corps des méthodes est implémenté pour faire fonctionner le service. Il déléguera les appels au service métier *GeographieMetierServiceImpl* indirectement via Spring.

3.3.2. ETAPE 2 (SERVEUR) : PARAMETRAGE DES PROVIDERS D'AUTHENTIFICATION

Contrairement aux configurations de base de Spring Security utilisant les notions de filtres pour gérer les aspects de sécurité, nous appliquerons une solution simplifiée. En effet, ces paramétrages de filtres sont utilisés surtout par les applications web dont la finalité est la vérification des différentes URLs appelées par les appelants. Ce n'est pas le cas étudié. Quels sont les paramétrages à effectuer pour les providers d'authentification ?

3.3.2.1. Qu'est ce que Spring AOP ?

La Programmation Orientée Aspect (POA, en anglais aspect-oriented programming - AOP) est un paradigme de programmation qui permet de séparer les considérations techniques (aspect en anglais) des descriptions métier dans une application. Empêcher un consommateur d'accéder à une méthode spécifique d'un Web Service, de manière classique, s'avère pari impossible, mais rendu possible via l'utilisation de l'AOP.

Spring Security s'intègre aux spécifications de l'AOP Alliance (implémentée en particulier par Spring AOP) et au Framework *AspectJ*. Leur utilisation étant très similaire, nous nous focaliserons particulièrement sur l'AOP Alliance, dont la gestion de la sécurité est très proche de ce qui se fait traditionnellement sur les transactions. Dans les deux cas, il s'agit d'ajouter un proxy sur les beans Spring et de modifier, par le biais de l'AOP, le comportement de certaines méthodes. L'une des possibilités offertes est le ***MethodSecurityInterceptor***.

3.3.2.2. Le fichier de contexte Spring

Un nouveau fichier de configuration Spring dénommé ***MyGeographie-SpringContext.xml*** a été créé à la racine du répertoire ***src*** du projet Apogée. Dans ce fichier de configuration Spring, la manière de configurer Spring Security est abordée. Nous devrons configurer le bean de la logique métier pour que

chaque invocation de ses méthodes soit interceptée. Cette interception se fait par le biais d'un bean : le ***securityInterceptor***. Le fragment de code du fichier de configuration de Spring suivant expose le paramétrage du bean ***securityInterceptor***. Il est fourni par la classe de Spring Security ***MethodSecurityInterceptor*** du package *org.springframework.security.intercept.method.aopalliance*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:sec="http://www.springframework.org/schema/security"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springframework.org/schema/security
                           http://www.springframework.org/schema/security/spring-security-2.0.xsd">

       <!-- ## ###################### ## -->
       <!-- ## geographieMetierService ## -->
       <!-- ## ###################### ## -->

<!-- Définition du bean geographieMetierService-->
<bean id="geographieMetierService"
       <!-- Définition de la classe de l'intercepteur transactionnel TrasactionProxyFactoryBean-->
       class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
              scope="singleton">
       <!-- Définition du manager des transactions-->
       <property name="transactionManager">
              <ref bean="transactionManager" />
       </property>
       <!-- Référence vers le target du bean geographieMetierService-->
       <property name="target">
              <ref local="geographieMetierServiceTarget" />
       </property>
       <!-- Définition des attributes de transaction-->
       <property name="transactionAttributes">
              <props>
                     <!-- Toutes les méthodes debutant par recuperer nécessitent une transaction-->
                     <prop key="recuperer*">PROPAGATION_REQUIRED, readOnly</prop>
              </props>
       </property>
       <!-- Définition de l'interceptor dans un bean securityInterceptor-->
       <property name="preInterceptors">
              <list>
                     <ref bean="securityInterceptor"/>
              </list>
       </property>
</bean>

<bean id="securityInterceptor"
       class="org.springframework.security.intercept.method.aopalliance.MethodSecurityInterceptor">
       <property name="authenticationManager"
              ref="authenticationManager" />
       <property name="accessDecisionManager"
              ref="accessDecisionManager" />
       <property name="objectDefinitionSource">
              <value>
                     gouv.education.apogee.commun.servicesmetiers.
                     GeographieMetierServiceInterface.recupererPays=ROLE_MANAGER
              </value>
       </property>
</bean>
<!-- Définition du provider d'authentication nommé authenticationManager-->
<bean id="authenticationManager"
       class="org.springframework.security.providers.ProviderManager">
       <property name="providers">
              <list>
                     <bean class="org.springframework.security.providers.
                            anonymous.AnonymousAuthenticationProvider">
                            <property name="key" value="anonymousKey" />
                     </bean>
              </list>
       </property>
</bean>
```

```
<bean id="accessDecisionManager" class="org.springframework.security.vote.UnanimousBased">
        <property name="decisionVoters">
                <list>
                        <bean class="org.springframework.security.vote.RoleVoter" />
                </list>
        </property>
</bean>
<!--Définition de la classe d'implémentation du bean geographieMetierService-->
<bean id="geographieMetierServiceTarget"
        class="gouv.education.apogee.commun.servicesmetiers.impl.GeographieMetierServiceImpl">
        <!— Cette propriété a été commentée pour éviter de charger le bean referentielGeographieService
        dont nous n'aurons pas besoin dans notre étude
                <property name="referentielGeographieService">
                        <ref bean="referentielGeographieService" />
                </property>
        -->
</bean>
<!--Définition du bean fournissant l'implémentation de l'interface UserManager-->
<bean id="userManager"
        class="gouv.education.apogee.commun.securite.springsecurity.UserManagerImpl">
</bean>

</beans>
```

Listing 3 – Un extrait du fichier de contexte Spring : MyGeographie-SpringContext.xml.

3.3.2.3. Le câblage

L'utilité de Spring pour relier tout l'ensemble réside dans le petit objet sur la figure présentant les modules, appelé AOP Proxy, qu'il génère. Ainsi, le premier bean : *geographieMetierServiceTarget*, permet de déclarer la classe *GeographieMetierServiceImpl*. Le second expose comment mettre en place la technique de l'AOP Proxy. Nous avons déclaré un bean *geographieMetierService* qui sera obtenu auprès de la factory **TransactionProxyFactoryBean**. Pour tout accès à ce bean, la factory retournera une instance de l'objet AOP Proxy. Avec la troisième propriété nommée **preInterceptors**, nous avons déclaré le bean *securityInterceptor* (explicité dans paragraphe suivant) pouvant intercepter les invocations de méthodes pour effectuer les contrôles de sécurité. Ceci nous permet de nous brancher dans les mécanismes de sécurité de Spring Security, sans aucune dépendance à la logique métier. Finalement, le bean *geographieMetierServiceTarget* spécifie la cible à l'implémentation réelle pour la propagation des invocations de méthodes à la classe *GeographieMetierServiceImpl*.

Notons par ailleurs que le bean « *userManager* » permet de se raccorder à la classe d'implémentation de l'interface *UserManager*, classe de gestion de l'entité désirant consommer le Web Service.

Le bean ***securityInterceptor*** a été configuré avec une propriété *authenticationManager* pour spécifier quel type d'authentification sera utilisé. Dans notre cas, notre conception repose sur une authentification basée sur l'Axis Handler, nous n'avons donc pas besoin d'utiliser les chaînes d'authentification propres à Spring Security. Par conséquent, nous allons la configurer avec l'**AnonymousAuthenticationProvider** (authentification anonyme), mode d'authentification permettant d'autoriser certaines ressources aux entités non authentifiées explicitement. Ainsi :

- Dans le cas où l'entité ne serait pas déclarée dans le fichier des entités (*users.xml*), on lui permet néanmoins d'accéder de façon anonyme aux autres méthodes non protégées de la classe.
- Dans le cas où l'entité serait déclarée, c'est-à-dire authentifiée dans le fichier des entités (*users.xml*), l'**AnonymousAuthenticationProvider** n'est pas sollicité.

Ceci nous permet ainsi de mettre en place notre propre processus d'authentification en utilisant le *SpringSecurityBridgeAuthenticationHandler* .

3.3.2.4. Un basic handler customisé

Les besoins en sécurité, nous poussent à avoir recours à cette classe. Dans notre exemple, il s'agit de la sécurité à deux niveaux différents : la sécurité au niveau du Web Service et la sécurité pour le code de l'application, c'est-à-dire les POJO. Axis nous donne la possibilité de nous brancher sur les Handlers customisés qui intercepteront les requêtes et les réponses afin de fournir une fonctionnalité supplémentaire qu'est l'authentification. Ainsi, il sera créé un Handler customisé appelé *SpringSecurityBridgeAuthenticationHandler* responsable de la sécurité du Web Service. Il étend la classe *BasicHandler*. Spring Security sera utilisé pour fournir un niveau de sécurité à l'application et plus spécialement aux POJO.

```
public class SpringSecurityBridgeAuthenticationHandler extends BasicHandler {
	/**
	 * Méthode appelée lors de l'envoi d'une requête vers le web service.
	 * @param msgContext
	 *            Le message contexte
	 * @exception AxisFault
	 *                Une faute Axis est levée
	 */
	public void invoke(MessageContext msgContext) throws AxisFault {
		// Récupération du bean userManager défini dans le contexte Spring
		UserManager userManager = (UserManager) SpringContext.getInstance()
				.getCtx().getBean("userManager");

		// login de l'entité et récupération d'un objet User
		User user = userManager.login(msgContext.getUsername(), msgContext
				.getPassword());

		// Dans le cas où l'entité existe et dispose de rôles
		if (user != null) {
			Authentication authOK = null;
			// Dans le cas où l'entité dispose de ROLES
			if (user.getCredentials() != null) {
				// Création d'une liste de rôles
				GrantedAuthority[] roles = new GrantedAuthority[user
						.getCredentials().length];
				for (int i = 0; i < user.getCredentials().length; i++) {
					// Création d'un objet rôle de type GrantedAuthorityImpl
					GrantedAuthorityImpl role = new GrantedAuthorityImpl(user
							.getCredentials()[i]);
					roles[i] = role;
				}
				// Création d'un token d'authentification avec des credentials
				authOK = new UsernamePasswordAuthenticationToken(user
						.getlogin(), user.getPassword(), roles);
			} else {
				// Dans le cas où l'entité ne dispose pas de ROLES
				// Création d'un token d'authentification sans credentials
				authOK = new UsernamePasswordAuthenticationToken(user
						.getlogin(), user.getPassword(), null);
			}
			// Insertion du token dans l'objet SecurityContextHolder
			SecurityContextHolder.getContext().setAuthentication(authOK);
		} else {
			// Dans le cas où l'entité n'est pas reconnue dans le fichier users.xml
			// lors de l'authentification, on crée alors une entité anonyme pour permettre
			// d'accéder aux ressources non protégées.
			Authentication authAnonymous = new AnonymousAuthenticationToken(
					"anonymousKey", "anonymous",
					new GrantedAuthority[] { new GrantedAuthorityImpl(
							"ROLE_ANONYMOUS") });
			// Insertion du token dans l'objet SecurityContextHolder
			SecurityContextHolder.getContext().setAuthentication(authAnonymous);
		}

	}
}
```

Listing 4 – Un extrait de la classe SpringSecurityBridgeAuthenticationHandler.java.

La création d'un *UsernamePasswordAuthenticationToken* fait suite à la récupération des informations d'authentification. Ces informations d'authentification, outre le username et le password, comprennent cependant, les autorisations, permissions accordées à cette entité. Ceci est possible grâce à l'interface *GrantedAuthority* et son implémentation *GrantedAuthorityImpl* du package *org.springframework.security*.

Dans le cas où les informations d'authentification ne pourraient pas être récupérées, on assiste à la création d'un *AnonymousAuthenticationToken*. Il permet de désigner une entité authentifiée de façon anonyme, ce qui lui permet d'accéder aux ressources non protégées.

3.3.3. ETAPE 3 (SERVEUR) : MISE EN OEUVRE DES INTERCEPTIONS POUR LES AUTORISATIONS

Ici, nous allons aborder la notion d'autorisation. Dans Spring Security, la gestion des autorisations est un mécanisme très évolué et, de 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 moindres frais. Pour cela, nous avons besoin de configurer le bean *securityInterceptor* avec la propriété **accessDecisionManager** pour spécifier comment il sera décidé d'accorder ou non l'accès à un appelant pour invoquer une méthode donnée.

Dans Spring Security, la permission d'accéder à une ressource est décidée par un système de vote. Cette ressource peut être protégée de 2 manières : par la sécurisation des URL l'utilisant ou par AOP. Nous nous pencherons sur la seconde possibilité. Afin de déterminer si un appelant a le droit d'accéder à une ressource, Spring Security utilise un système de vote grâce à la classe essentielle : *org.springframework.security.vote.rolevoter*. Cette classe valide l'accès à une ressource en fonction des rôles possédés par l'appelant. Elle se configure dans le fichier de configuration Spring *MyGeographie-SpringContext.xml*. Notons que cette classe ne vote que pour les ressources protégées par un rôle. Etant donné que plusieurs de ces beans de vote peuvent être utilisés pour protéger une ressource, plusieurs votes sont alors émis pour protéger l'accès à une ressource donnée.

```
<bean id="geographieMetierService"
	class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
		scope="singleton">
	<property name="transactionManager">
		 <ref bean="transactionManager" />
	</property>
	<property name="target">
		<ref local="geographieMetierServiceTarget" />
	</property>
	<property name="transactionAttributes">
		<props>
			<prop key="recuperer*">PROPAGATION_REQUIRED, readOnly</prop>
		</props>
	</property>
	à Définition de l'interceptor dans un bean securityInterceptor
	<property name="preInterceptors">
		<list>
			<ref bean="securityInterceptor"/>
		</list>
	</property>
</bean>

<bean id="securityInterceptor"
	class="org.springframework.security.intercept.method.aopalliance.MethodSecurityInterceptor">
	<property name="authenticationManager"
```

```
            ref="authenticationManager" />
        <property name="accessDecisionManager"
            ref="accessDecisionManager" />
```

```
        <!-- Définition de la propriété objectDefinitionSource : permissions d'accès à une méthode-->
        <property name="objectDefinitionSource">
            <value>
            <!--L'accès à la méthode recupererPays nécessitera le rôle ROLE_MANAGER-->
            gouv.education.apogee.commun.servicesmetiers.
            GeographieMetierServiceInterface.recupererPays=ROLE_MANAGER
            </value>
        </property>
```

```
</bean>
<bean id="authenticationManager" class="org.springframework.security.providers.ProviderManager">
        <property name="providers">
                <list>
                        <bean class="org.springframework.security.providers.
                                anonymous.AnonymousAuthenticationProvider">
                                <property name="key" value="changeThis" />
                        </bean>
                </list>
        </property>
</bean>
```

```
<!--Définition du bean accessDecisionManager-->
<bean id="accessDecisionManager"
        class="org.springframework.security.vote.UnanimousBased">
        <!--Définition du type de voter ici RoleVoter-->
        <property name="decisionVoters">
                <list>
                        <bean class="org.springframework.security.vote.RoleVoter" />
                </list>
        </property>
</bean>
```

```
<bean id="geographieMetierServiceTarget"
        class="gouv.education.apogee.commun.servicesmetiers.impl.GeographieMetierServiceImpl">
        <!— Cette propriété a été commentée pour éviter de charger le bean referentielGeographieService
        dont nous n'aurons pas besoin dans notre étude
        <property name="referentielGeographieService">
                        <ref bean="referentielGeographieService" />
                </property>
        -->
</bean>
<bean id="userManager"
        class="gouv.education.apogee.commun.securite.springsecurity.UserManagerImpl">
</bean>

</beans>
```

Listing 5 – Extrait du fichier MyGeographie-SpringContext.xml : Les attributs pour les autorisations.

Le traitement des votes est effectué par un autre bean, utilisant l'interface *org.springframework.security.vote.accessDecisionVoter*. Il en existe 3 implémentations différentes, en fonction de la manière dont on souhaite traiter les votes. Elles sont contenues dans le package *org.springframework.security.vote* :

- *affirmativeBased* : donne l'autorisation d'accéder à la ressource, si l'un au moins des votants a donné son accord ;
- *conscensusBased* : donne l'autorisation si la majorité des votants est d'accord ;
- *unanimousBased* : donne l'autorisation uniquement si l'ensemble des votants est d'accord.

Maintenant, au niveau des « voters », Spring Security 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 (Access Control List) : READ, WRITE, DELETE,...

Nous avons choisi d'implémenter le manager de type *UnanimousBased* et comme électeur (*decisionVoter*) le *RoleVoter*. Le paramétrage de ces différents éléments se trouve dans le listing de code précédent.

La dernière propriété à configurer est l'*ObjectDefinitionSource* qui permet de spécifier quelles permissions sont requises pour accéder aux différentes méthodes de la classe à sécuriser. Dans cette implémentation, l'objectif est de sécuriser la méthode *recupererPays()* et d'en accorder l'accès aux entités ayant le rôle : *ROLE_MANAGER*. Ceci se fait en précisant le chemin complet d'accès à la méthode.

Notons aussi qu'il est possible de définir un même rôle par exemple pour l'ensemble des méthodes commençant par une chaîne de caractères précise et appartenant à une même classe, en utilisant le « wildcard » (*):

```
<!-- Définition de la propriété objectDefinitionSource : permissions d'accès à une (des) méthode (s) -->
<property name="objectDefinitionSource">
        <value>
        <!--L'accès à toutes les méthodes commençant par recuperer nécessitera le rôle
        ROLE_MANAGER-->
        gouv.education.apogee.commun.servicesmetiers.
        GeographieMetierServiceInterface.recuperer*=ROLE_MANAGER
        </value>
</property>
```

Listing 6 – Extrait du fichier MyGeographie-SpringContext.xml : Le wildcard (*) pour la définition des rôles.

3.3.3.1. Un intercepteur : Le MethodSecurityInterceptor

Comme son nom l'indique, cette classe est utilisée pour appliquer la sécurité en interceptant les invocations de méthodes dans un premier temps, et en vérifiant que les appelants sont authentifiés et autorisés, dans un second temps. Ainsi son fonctionnement s'axe autour de 4 étapes :

1. Il vérifie les sources de définition des objets pour s'assurer que le proxy est un objet sécurisé. Si c'est le cas, alors on passe à l'étape 2.
2. Il vérifie qu'un objet d'authentification est inséré dans le *SecurityContextHolder*. Si un tel objet est inexistant, alors une exception du type *AuthenticationCredentialsNotFoundException* est levée. Si par contre un tel objet existe, il s'assurera qu'il a été authentifié. Si ce n'est pas le cas, nous nous rendons alors à l'étape 3, sinon, à l'étape 4.
3. Il essayera de procéder à l'authentification en utilisant *l'authenticationManager*. Il est très important ici de cerner le fait que *l'AnonymousAuthenticationProvider,* défini dans le fichier Spring, n'est utilisé que dans le cas où l'authentification aurait échoué en amont. Dans ce cas, il se charge d'authentifier l'entité. Si l'authentification se déroule avec succès, on aboutit à l'étape 4, sinon une exception *AuthenticationException* est levée.
4. Il utilise l'*AccessDecisionManager* pour décider si l'entité authentifiée dispose de droits suffisants pour atteindre la ressource. Si ce n'est pas le cas, une exception de type *AccessDeniedException* est alors levée, sinon l'invocation de la méthode pourra se dérouler.

3.3.3.2. Le câblage

Grâce à l'intercepteur *MethodSecurityInterceptor*, nous pouvons nous assurer que l'entité consommatrice est bien authentifiée et autorisée à accéder à la ressource désirée. Puisque nous avons eu recours au *RoleVoter* pour statuer sur les décisions d'accès, les permissions accordées sont alors des rôles (ROLE_). Dans notre étude, il faudrait alors disposer du rôle *ROLE_MANAGER* pour invoquer la méthode protégée.

Les rôles associés à l'entité souhaitant consommer le Web Service, sont récupérés en même temps que lors de la vérification de sa chaîne username/password dans un fichier XML à plat dans notre cas.

Une instance de *UsernamePasswordAuthenticationToken* est enfin créée, en lui passant comme paramètres : username, password et les rôles.

```
// Création d'un token d'authentification avec des credentials
authOK = new UsernamePasswordAuthenticationToken(user.getlogin(), user.getPassword(), roles);
// Insertion du token dans l'objet SecurityContextHolder
SecurityContextHolder.getContext().setAuthentication(authOK);
```

Listing 7 – Extrait du fichier SpringSecurityBridgeAuthenticationHandler.java : Création du UsernamePasswordAuthenticationToken.

En utilisant ce constructeur acceptant un tableau de rôles de type *GrantedAuthority*, nous informons ainsi Spring Security que ce token (jeton) est effectivement authentifié, ce qui le dispense réellement d'une nouvelle authentification. Enfin, nous insérons le jeton dans le *SecurityContext* par le biais de la méthode static *getContext()* sur *SecurityContextHolder* et *setAuthentication(Authentication authentication)*.

Une instance de *AnonymousAuthenticationToken* est créée dans le cas où l'authentification n'a pas pu s'opérer. Il comporte une clef « *anonymousKey* », une entité (« principal ») « *anonymous* » et un rôle « *ROLE_ANONYMOUS* ». Nous insérons ce token (jeton) dans le *SecurityContextHolder*, ce qui permet au provider de l'*authenticationManager*, *AnonymousAuthenticationProvider*, d'estimer authentifiée l'entité contenue dans le token.

Cette entité anonyme peut alors accéder aux autres ressources (méthodes) non explicitement protégées.

```
// Dans le cas où l'entité n'est pas reconnue dans le fichier users.xml
// lors de l'authentification, on crée alors une entité anonyme pour permettre
// d'accéder aux ressources non protégées.
Authentication authNOK = new AnonymousAuthenticationToken( "anonymousKey", "anonymous",
              new GrantedAuthority[] { new GrantedAuthorityImpl("ROLE_ANONYMOUS") });
// Insertion du token dans l'objet SecurityContextHolder
```

Listing 8 – Extrait du fichier SpringSecurityBridgeAuthenticationHandler.java : Création de l'AnonymousAuthenticationToken.

3.3.4. ETAPE 4 (SERVEUR) : LE DESCRIPTEUR DE DEPLOIEMENT : SERVER-CONFIG.XML

Afin d'utiliser des fonctionnalités avancées en terme de déploiement, il est alors possible d'utiliser les descripteurs de déploiement. Ce principe implémenté par Axis permet de définir les méthodes que nous

souhaitons publier dans le Web Service, le portType, l'encodage… Ceci est représenté au sein d'un fichier : *le server-config.xml*. Il s'agit d'un fichier XML qui répond à une grammaire précise située dans le répertoire WEB-INF.

Ce fichier constitue l'un des principaux connecteurs dans notre procédure de sécurisation. Il permet de définir des Handlers spécifiques à la fois pour les messages entrants dits « *requestFlow* » et les messages sortants : « *responseFlow* ». Cette technique nous a donc permis de spécifier notre classe Handler customisée : *SpringSecurityBridgeAuthenticationHandler* qui est alors exécutée pour toutes les requêtes entrantes vers le Web Service. Il peut être spécifié ou non pour chacun des services.

```
<service name="GeographieMetier" provider="java:RPC" style="wrapped" use="literal">
...
        <requestFlow>
                <handler type="java:gouv.education.apogee.commun.securite.springsecurity.
                                                SpringSecurityBridgeAuthenticationHandler"/>
        </requestFlow>
...
</service>
```

Listing 9 - Un extrait du fichier server-config.wsdd

Nous avons montré plus haut dans ce document que le Handler customisé « récupère » la chaîne usersname/password depuis le MessageContext Axis. Il était malgré tout judicieux d'illustrer le mécanisme par lequel Axis insérait ces données dans cet objet MessageContext à partir du client. En effet, afin d'acheminer la chaîne de connexion usermane/password, nous nous sommes basés sur le modèle de sécurité intégrée simple (Simple Security Provider). Il s'agit du modèle par défaut dans Axis. Dans ce modèle, les informations de sécurité (« Security Credentials ») de l'appelant doivent être récupérées à partir des informations d'en-tête http. A cet effet, nous avons déployé un gestionnaire de message spécifique réalisant cette opération : **org.apache.axis.handlers.http.HTTPAuthHandler,** sur le serveur Axis (fichier *server-config.wsdd*) dans la chaîne de requête de transport, pour le protocole http.

```
<transport name="http">
        <requestFlow>
                <handler type="URLMapper" />
                <!--Récupération des informations username/password de l'entité dans l'en-tête http-->
                <handler type="java:org.apache.axis.handlers.http.HTTPAuthHandler"/>
        </requestFlow>
        <!--Pas de query string possible (?wsdl sur les services) -->
        <parameter name="useDefaultQueryStrings" value="false" />
</transport>
```

Listing 10 - extrait de server-config.wsdd : récupération du username/password

3.3.5. ETAPE 5 (SERVEUR) : READAPTATION DE L'EXISTANT

Appliquer de la sécurité à une application existante n'est pas toujours sans conséquences sur l'environnement d'exécution tant différents modules sont appelés à collaborer ensemble. Ceci peut impliquer des dysfonctionnements entre par exemple les librairies existantes et celles à rajouter. De même, les paramétrages des fichiers de configuration peuvent aussi évoluer pour supporter les notions de sécurisation. Il s'avère alors nécessaire de réadapter l'existant afin d'y intégrer les nouveaux paramétrages.

3.3.5.1. La remontée des exceptions

Lors des échanges entre le client et le Web Service géographie, des erreurs concernant les notions de sécurité (Authentification, Autorisations) peuvent être remontées sous formes d'exceptions au client. Cependant, ces exceptions, si elles ne sont pas réadaptées, ne pourront pas être compréhensibles par le client. Ainsi, lors d'une génération d'exception au niveau du fournisseur, il a été nécessaire de retravailler cette dernière avant sa propagation.

La classe *gouv.education.apogee.commun.transverse.utils.ExceptionConverter.java* a donc été modifiée pour retransmettre les exceptions de types :

- *AuthenticationCredentialsNotFoundException* : dans le cas où l'entité ne serait pas authentifiée (inexistence d'un objet *Authentication* dans le *SecurityContextHolder*).
- *AccessDeniedException* : dans le cas où l'entité authentifiée, n'aurait pas les privilèges nécessaires pour exécuter une méthode donnée.

```
public class ExceptionConverter {
	/**
	 * Convertit tous les types d'exceptions en WebBaseException qui pourra être
	 * serialisée par Web Service en tant qu'élément wsdl:fault
	 * @param _ex
	 *            Exception à transformer
	 * @param _nomWebService
	 *            Nom du web service ayant lancé l'exception
	 *
	 * @return WebBaseException l'exception d'entrée transformée en exception
	 *         Web
	 */
	public static WebBaseException convertirException(Exception _ex,
			String _nomWebService) {
		// ACB : on logue car sinon pas de traces en WS
		_ex.printStackTrace();
		if (_ex instanceof WebBaseException) {
			// Les exceptions de sous-type WebBaseException sont
			// transformées en WebBaseException de haut niveau pour pouvoir être
			// envoyées sous forme de wsdl:fault à l'appelant
			return new WebBaseException((WebBaseException) _ex);
		}
		// Problème de transaction bdd
		else if (_ex instanceof TransactionException) {
			WebDataException tmp = null;
			// Recherche de la cause root
			// On pourrait utiliser getRootCause() mais dispo que sur Spring 2.0
			Throwable rootCause = _ex;
			while (true) {
				rootCause = rootCause.getCause();
				// Si pas de cause fils, on sort
				if (rootCause == null) {
```

```
                                        break;
                                }
                                // Si on trouve une SQLException, on sort
                                if (rootCause instanceof SQLException) {
                                        break;
                                }
                        }
                        // Si pb d'acces à la base
                        if ((rootCause != null) && rootCause instanceof SQLException) {
                                int errorCode = ((SQLException) rootCause).getErrorCode();
                                tmp = new WebDataException("transaction", "connexion",
                                                "Problème de connexion à la base de données (code oracle "
                                                        + errorCode + ") pour le service "
                                                        + _nomWebService, _ex);
                        }
                        // Autre type de pb de transaction
                        else {
                                tmp = new WebDataException("transaction", "generique",
                                "Problème de transaction générique à la base de données pour le service "
                                                        + _nomWebService, _ex);
                        }
                        return new WebBaseException(tmp);
                }
                // Problème d'acces aux données
                else if (_ex instanceof DataAccessException) {
                        return new WebBaseException(new WebDataException("dao", "",
                                "Probleme d'accès aux données pour le service " + _nomWebService, _ex));
```

```
                // Dans le cas où il s'agit d'un client non authentifié
                } else if (_ex instanceof AuthenticationCredentialsNotFoundException) {
                        return new WebBaseException(
                                new WebBaseException("authentification",
                                "Authentification : Le client n'est pas authentifié ou n'est pas
                                        déclaré pour exécuter la méthode " + _nomWebService, _ex));
                        // Dans le cas où il s'agit d'un accès refusé pour manque de
                        // privilèges
                } else if (_ex instanceof AccessDeniedException) {
                        return new WebBaseException(
                                new WebBaseException("authentification",
                                "Accès refusé : Le client ne dispose pas de privilèges suffisants
                                        pour exécuter la méthode " + _nomWebService, _ex));
                }
```

```
                // Dans tous les autres cas d'exception -> retour d'une exception
                // générique
                else {
                        return new WebBaseException("webservice",
                                        "Probleme lors de l'appel au service " + _nomWebService,
                                        _ex);
                }
        }
}
```

Listing 11 - Un extrait du fichier ExceptionConverter.java : Le traitement des exceptions d'authentification.

3.3.5.2. Montée en version ?

L'utilisation du Framework Spring Security dans sa version 2.0.3 s'avère incompatible avec la version du Framework Spring 2.0.1 utilisée dans les Web Services Apogée. Nous avons dû monter en version le Framework Spring à 2.0.8 afin de les faire fonctionner ensemble.

3.3.5.3. Conséquences : Réadaptation de l'intercepteur transactionnel

La définition de la classe du *TrasactionProxyFactoryBean* a été aussi revue pour s'adapter avec l'attribut *singleton*.

```
<bean id="geographieMetierService"
              class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
       singleton="true" >
```

Listing 12 - Un extrait du fichier MyGeographie-SpringContext.java : L'intercepteur transactionnel (Avant).

Si rien n'est spécifié, Spring considère tout bean comme un singleton. Autrement dit, lorsque deux demandes sont faites pour un même bean, c'est la même instance de l'objet qui est fournie par le BeanFactory/ApplicationContext. Ceci dit, pour se conformer à la définition de base de l'existant via la version 2.0.8 de Spring, c'est au travers de l'attribut Scope que cela est géré :

```
<bean id="geographieMetierService"
              class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
       scope="singleton">
```

Listing 13 - Un extrait du fichier MyGeographie-SpringContext.java : L'intercepteur transactionnel (Après).

La montée en version du Framework Spring nous amène à redéfinir l'intercepteur transactionnel *TrasactionProxyFactoryBean* de Spring devant gérer les transactions au sein du service. Il s'est alors avéré important d'adapter à la fois la notion de transaction avec celle de sécurité et ce, pour le bean représentant le Web Service géographie. Ceci s'est opéré au niveau des propriétés transactionnelles du service. Pour ce type de déclaration, on a la possibilité de définir des « pré-intercepteurs » qui vont être appelés avant l'appel effectif de l'objet dit « target » (c'est-à-dire ici la classe *GeographieMetierServiceImpl*). Le bean *geographieMetierService* sera ainsi protégé par le *securityInterceptor* défini.

```
<bean id="geographieMetierService"
       <!-- Définition de la classe de l'intercepteur transactionnel TrasactionProxyFactoryBean-->
       class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
              scope="singleton">
       <property name="transactionManager">
               <ref bean="transactionManager" />
       </property>
       <property name="target">
              <ref local="geographieMetierServiceTarget" />
       </property>
       <!-- Définition des attributes de transaction-->
       <property name="transactionAttributes">
              <props>
                     <prop key="recuperer*">PROPAGATION_REQUIRED, readOnly</prop>
              </props>
       </property>
       <!-- Définition d'un pre-intercepteur dans un bean securityInterceptor-->
       <property name="preInterceptors">
              <list>
                     <ref bean="securityInterceptor"/>
              </list>
       </property>
</bean>
```

Listing 14 - Un extrait du fichier MyGeographie-SpringContext.xml : La définition des preInterceptors.

3.3.6. ETAPE 6 (CLIENT) : PARAMETRAGE DES APPELS

Pour permettre au client de dialoguer avec le Web Service il devra respecter la politique de consommation mise en place. Celle-ci se traduit en substance par le fait que ce dernier devra fournir des données supplémentaires : username/password pour authentification, en plus de sa requête SOAP. Cette chaîne de connexion servira aussi à récupérer ses permissions. Nous nous baserons sur la méthode *recupererPays()*, du fichier java : *TestWSGeographie.java* dans le package *testwsclient* du projet client Apogee *apo-testwsclient*.

Il en ressort alors que quelques légères modifications ont dû être apportées au client pour effectuer son appel. Il s'agira d'insérer un username de même qu'un password dans l'entête du protocole http devant véhiculer le message SOAP. Le listing suivant montre les modifications subies par le client.

```
//Instanciation d'un service locator
GeographieMetierServiceInterfaceServiceLocator sLocator = new
                                    GeographieMetierServiceInterfaceServiceLocator();
//Récupération du Stub (bouchon client) à partir du service géographie
GeographieMetierSoapBindingStub ws =  (GeographieMetierSoapBindingStub)
                                                    sLocator.getGeographieMetier();
//Insertion du username
ws.setUsername("myUser");
//Insertion du password
ws.setPassword("myPass");

//Exécution de la méthode recupererPays
pays = ws.recupererPays(_cod,_temoinEnService);
```

Listing 15 - Un extrait du fichier TestWSGeographie.java : L'appel client.

Dans notre exemple, nous appelons directement la classe GeographieMetierSoapBindingStub. Afin de s'intégrer au mieux avec l'architecture existante côté client, il serait bon d'étudier la possibilité d'étendre les fonctionnalités des proxys afin d'intégrer la possibilité de passer ou non un username/password. Ceci doit bien sûr être fait dans le respect des contraintes de l'architecture client.

3.4. « DEBRAYAGE » POUR UN RETOUR A L'ETAT INITIAL

Offrir une solution « débrayable », c'est-à-dire, la possibilité d'activer ou pas la sécurisation des Web Services Apogée était une des contraintes de notre implémentation. Ce « débrayage » doit s'effectuer en suivant les étapes suivantes :

- Commenter dans le fichier server-config.wsdd le Handler *SpringSecurityBridgeAuthenticationHandler*, principal connecteur des messages entrants dits « *requestFlow* » vers le Web Service. Ce Handler est exécuté pour toutes les requêtes vers le fournisseur de service. Il faudra donc commenter celui du service désiré, ici dans notre cas, celui spécifique au service géographie. Mis en commentaire, ce Handler sera donc inactif ce qui impliquera que tous les appels vers le service seront directement acheminés.

```
<service name="GeographieMetier" provider="java:RPC" style="wrapped" use="literal">
...
      <!-- requestFlow>
            <handler type="java:gouv.education.apogee.commun.securite.
                  springsecurity.SpringSecurityBridgeAuthenticationHandler"/>
      </requestFlow-->
...
</service>
```

Listing 16 - Un extrait du fichier server-config.wsdd : Le débrayage du Handler SpringSecurityBridgeAuthenticationHandler.

- Commenter dans le fichier *MyGeographie-SpringContext.xml* (ou le fichier de configuration Spring visé), la propriété définissant les « pre-interceptor » permettant de déclarer le bean *securityInterceptor*. Ceci empêchera ainsi de passer par la *MethodSecurityInterceptor* (pour vérifier les notions d'authentification et d'autorisation) avant d'exécuter la méthode visée.

```
<bean id="geographieMetierService"
        <!-- Définition de la classe de l'intercepteur transactionnel TrasactionProxyFactoryBean-->
        class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean"
                scope="singleton">
        <property name="transactionManager">
                 <ref bean="transactionManager" />
        </property>
        <property name="target">
                <ref local="geographieMetierServiceTarget" />
        </property>
        <!-- Définition des attributes de transaction-->
        <property name="transactionAttributes">
                <props>
                        <prop key="recuperer*">PROPAGATION_REQUIRED, readOnly</prop>
                </props>
        </property>
        <!-- Définition d'un pre-intercepteur dans un bean securityInterceptor-->
                <!-- property name="interceptorNames">
                        <list>
                                <value>securityInterceptor</value>
                        </list>
                </property-->
</bean>
```

Listing 17 - Un extrait du fichier MyGeographie-SpringContext.xml : Le débrayage de l'intercepteur securityInterceptor.

- De façon optionnelle, on peut aussi commenter dans le fichier web.xml les listeners *org.springframework.web.context.ContextLoaderListener* en charge d'écouter le chargement du contexte Spring et *gouv.education.apogee.commun.securite.springsecurity.utils.StartupListener*, listener customisé permettant d'écouter aussi le chargement du contexte Spring dans le but d'en récupérer l'instance. Ceci dit, laisser ces entrées en l'état (c'est-à-dire sans les commenter) n'aura aucune incidence sur le fonctionnement normal de l'application.

3.5. SYNTHESE : FONCTIONNEMENT PAR LA PRATIQUE

Pour faire fonctionner cet ensemble de manière transparente, nous décrivons dans cette section un diagramme de séquences afin de faire une synthèse sur les différents mécanismes.

3.5.1. LE TOUT EN 1 : UN DIAGRAMME DE SEQUENCES

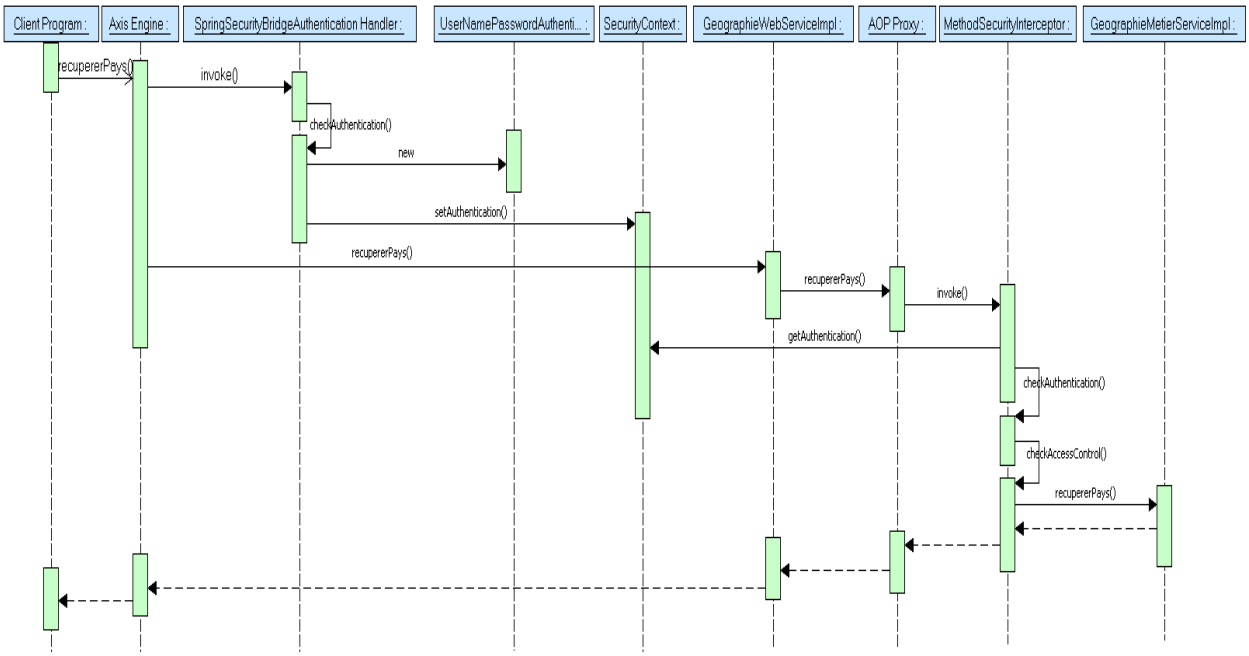

Figure 6 - Un diagramme de séquences des interactions entre les modules.

Le diagramme de séquences précédent montre aisément le flux des traitements qui se produit lorsqu'un client appelle le service *GeographieWebServiceInterface*. La requête est reçue par le moteur Axis (Axis Engine) qui à son tour invoque le pont personnalisé : *SpringSecurityBridgeAuthenticationHandler*. Ce dernier vérifie les informations d'authentification et crée un jeton *UsernamePasswordAuthenticationToken*. Ensuite, ce jeton est inséré dans le *SecurityContext* pour une utilisation par Spring Security par la suite. Lorsque les vérifications se soldent par un succès, le moteur Axis appelle la méthode *recupererPays()* de la classe *GeographieWebServiceImpl*. *GeographieWebServiceImpl* délègue alors cet appel au proxy AOP, instancié plus tôt durant l'initialisation du contexte web de Spring ***MyGeographie-SpringContext.xml***. Le proxy AOP appelle la classe *MethodSecurityInterceptor* de Spring Security pour qu'il puisse effectuer ses contrôles de sécurité. *MethodSecurityInterceptor* reçoit le jeton d'authentification de la *SecurityContext* et vérifie s'il a déjà été authentifié. Ensuite, il utilise les informations contenues dans le jeton d'authentification pour voir si le client peut avoir accès pour invoquer la méthode *recuperePays()* sur *GeographieMetierServiceImpl*. Si l'accès est autorisé au client, alors *MethodSecurityInterceptor* permet

l'invocation de la méthode. *GeographieMetierServiceImpl*, procède au traitement de la requête et propage la réponse vers le programme client.

3.5.2. LES FLUX ECHANGES

3.5.2.1. Appel autorisé et authentifié

La figure suivante expose les différents flux échangés entre le client et le Web Service. La première fenêtre correspond au message SOAP émis par le client vers le Web Service. Nous y voyons bien dans le corps du message, les paramètres d'appel de la méthode *recupererPays()*. Cet appel a été authentifié et autorisé. La seconde fenêtre correspond au message SOAP en guise de réponse acheminée du Web Service vers le client. On y voit les caractéristiques du pays retourné.

Ici Tcpmon écoute sur le port 8081 (port d'émission du message du client) et retransmet sur le port 8082, port d'écoute du Web Service. Notons que la chaîne *« Authorization : Basic bX1Vc2Vy0m15UGFzcw== »* correspond à username/password inséré par le client dans son appel. Ces données ont été encodées en Base68 et passées dans l'en-tête http, dans la propriété nommée *Authorization.* Dans le Handler *HTTPAuthHandler,* ces données sont décodées et insérées dans le messageContext comme étant les propriétés username et password.

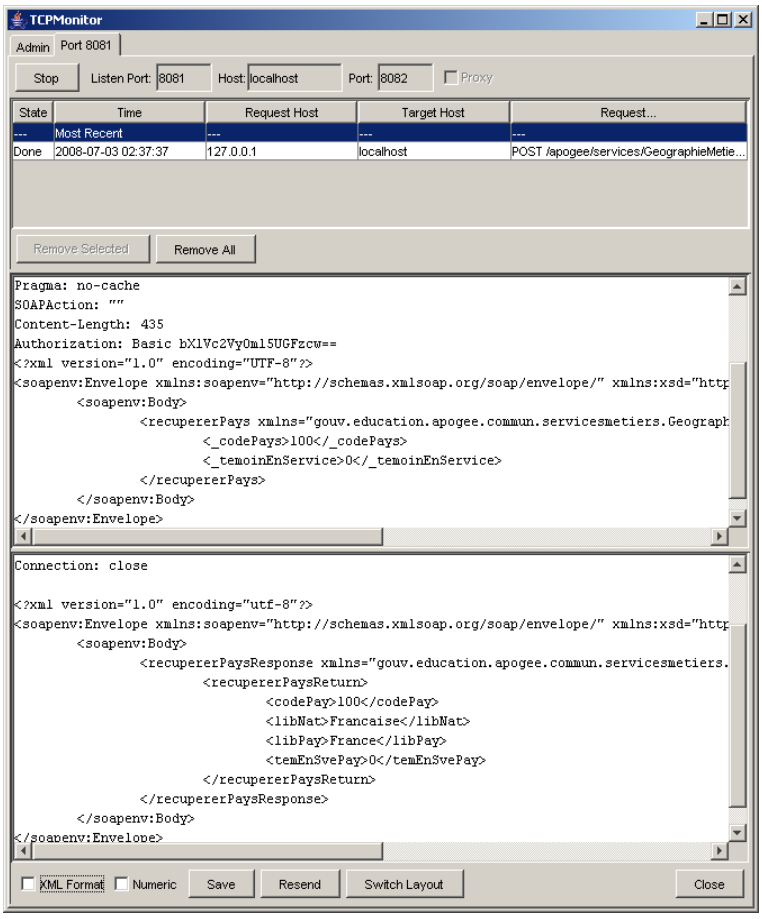

Figure 7 - Fenêtre TcpMon : Exemple d'un appel autorisé et authentifié.

3.5.2.2. Appel authentifié et non autorisé

La figure suivante expose les différents flux échangés entre le client et le Web Service. La première fenêtre correspond au message SOAP émis par le client vers le Web Service. Nous y voyons bien dans le corps du message, les paramètres d'appel de la méthode *recupererPays()*. Cet appel a été authentifié et non autorisé. Nous devons alors nous attendre à une erreur renvoyée par le Web Service. Ainsi, la seconde fenêtre correspond au message SOAP en guise de réponse acheminé du Web Service vers le client. On y voit bien qu'une exception a été remontée vers le client.

Ici Tcpmon écoute sur le port 8081 (port d'émission du message du client) et retransmet sur le port 8082, port d'écoute du Web Service.

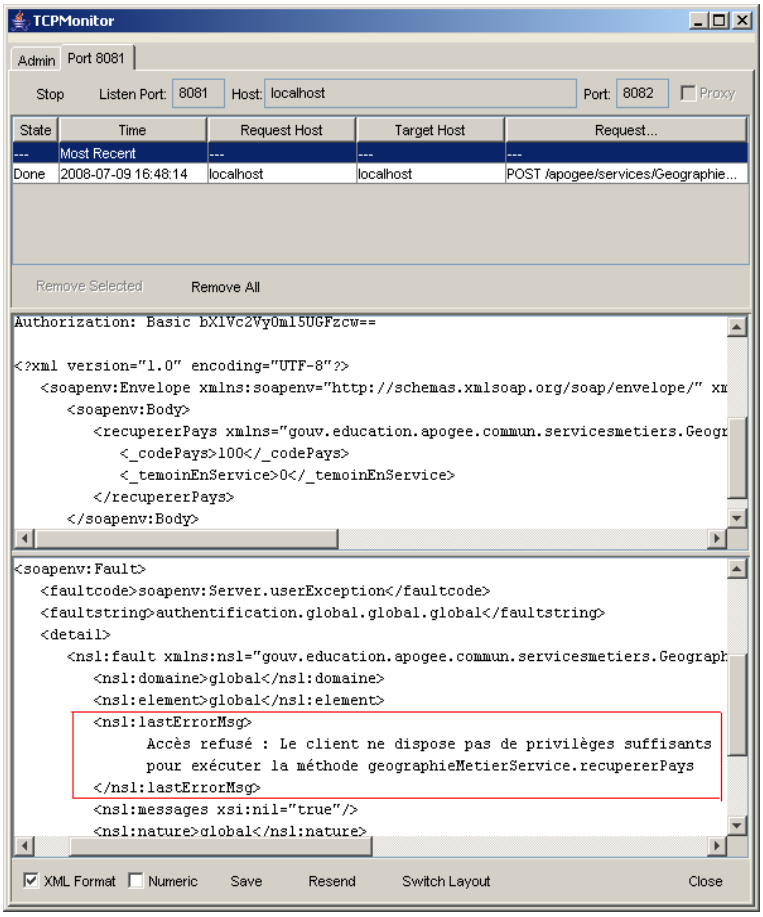

Figure 8 - Fenêtre TcpMon : Exemple d'un appel authentifié et non autorisé.

3.5.2.3. Appel non authentifié et non autorisé

Ce cas est similaire au précédent. Le client n'étant ni authentifié, ni autorisé, il n'existe donc aucun objet de type *Authentication* dans le *SecurityContextHolder*, le Web Service lui remonte alors une exception.

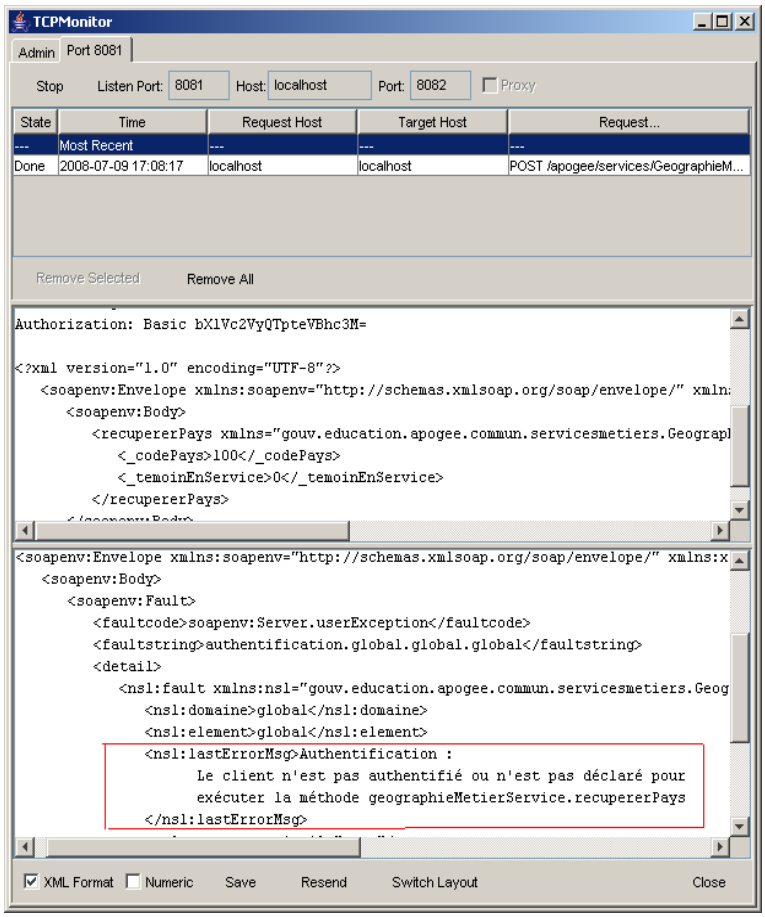

Figure 9 - Fenêtre TcpMon : Exemple d'un appel non authentifié et non autorisé.