

Déploiement d'un Web Service avec Spring WS

par Gildas Cuisinier ([Hikage](#)) ([Blog](#))

Date de publication : 10 Mars 2008

Dernière mise à jour :

Spring WS est un framework développé dans le but d'aider les développeurs à mettre en place de nouveaux services Web.

Mais qu'apporte-t-il de plus que les autres ? Quelles sont ces particularités ?

Cet article va tenter de vous présenter ce projet, ainsi qu'un cas pratique sur base de celui-ci.

I - Présentation

- Méthode : Code First

- Méthode : Contract First

- Spring WS

- Quand choisir Spring WS et pourquoi ?

 - Evolution

II - Cas pratique

- Déploiement du service

 - Création du projet

 - Création du contrat

 - Implémentation de l'Endpoint

 - Ajout de dépendances

 - Création de l'endpoint

 - Routage des messages vers le Endpoint

 - Publication du WSDL

 - Test du service avec SoapUI

- Création d'un client

- Améliorations du services

 - Validation

 - WS-Security

III - Conclusion

- Remerciements

I - Présentation



Comme le sous-entend son nom, le projet Spring WS est un framework dédié au développement de services Web. Il vient donc s'ajouter dans une liste déjà conséquente d'API/Framework Java : Axis, XFire, JAX-WS, JBoss WS, ...

Mais quels sont donc les avantages de Spring WS par rapport à ces autres framework ? Avant de répondre à cette question, et de présenter plus en détails ce projet, il est intéressant de rappeler quelles sont les deux manières de créer un service Web.

Méthode : Code First

La méthode la plus populaire est sans doute la méthode Code-First (Contract-Last, Bottom-Up.), celle-ci est en effet la plus simple à mettre en #uvre.

Le principe est de créer le service sous forme de classes (et d'interfaces) java, et d'utiliser un outil (généralement inclus dans l'API utilisé) qui permet de générer automatiquement le WSDL correspondant.

Cette étape peut prendre différentes formes : génération manuelle du WSDL grâce à un outil, génération automatique grâce une fabrique Spring, ou encore génération automatique à partir d'annotations Java 5.

Cette manière de procéder ne nécessite pas spécialement une grande connaissance de SOAP, des WSDL, voir même de XML. De plus, le déploiement d'un tel service peut être très rapide : une classe, quelques annotations, un serveur JEE compatible et voilà un service Web tout fait.

En contre partie, ne pas avoir la main sur le WSDL peut être gênant. Une simple modification d'une classe Java utilisée par le WebService peut provoquer la génération d'un WSDL complètement différent, conséquence de cela, les clients créés à partir de la précédente version ne fonctionneront plus.

Ceux-ci devront donc générer à nouveau leur classe d'accès au service web à partir du nouveau WSDL.

De même, si pour une raison ou l'autre l'API que vous avez choisie n'est plus maintenue, le WSDL généré par une autre API ne sera pas obligatoirement identique, et aura les mêmes conséquences que le point précédent.

Par ailleurs, le WSDL généré est parfois très "bavard" et loin d'être optimisé.

Méthode : Contract First

L'autre méthode est la méthode Contract-First (ou Top-Down). Dans cette méthode, l'important est le contrat du service, l'implémentation étant secondaire.

La première étape est donc de définir son propre WSDL : types, messages, ports, binding, services.


Une connaissance de XML, de SOAP et du WSDL est donc absolument nécessaire.

Une fois le contrat créé, il suffit de créer l'implémentation à partir des outils proposés par les API (Script Ant, exécutable, plugin dans IDE, ..).

Le premier avantage de cette méthode est que l'on n'est pas fortement lié à un framework. Le WSDL étant fixé, le changement d'API coté serveur est totalement transparent pour les clients du service.

A fortiori, le service n'est pas lié à un langage en particulier, il serait tout à fait possible de remplacer une implémentation Java par une implémentation en C# sans affecter les clients.

Le second avantage, selon moi, est que cela force à mieux découper son application en couches. D'un coté une couche **service** générale qui s'occupe de la partie métier, des transactions, possède ses propres objets métiers. De l'autre une couche **service Web**, qui va utiliser la couche générale, mais qui possèdera ses propres DTO.

 *Il est bien évidemment possible de faire une découpe de ce type dans un service web code-first. Mais le fait de pouvoir déployer un service existant en l'annotant permet de se détourner de cette bonne pratique.*

Spring WS

Spring WS de son coté ne permet qu'une seule méthode : Contract First, et de plus, il ne gère que les services orientés Document (par opposition au service RPC).

Il faut savoir que dans un service Web orienté Document, c'est la définition des messages échangés qui importe le plus. Et c'est dans cette optique que Spring WS a été créé.

En effet, Spring WS n'oblige pas le développeur à fournir un fichier WSDL complet (même si l'option reste possible), celui-ci étant assez complexe.

La seule information demandée par Spring WS est une définition des messages échangés, et cela sous la forme d'un schéma XML.

Quand choisir Spring WS et pourquoi ?

Quand ?

De part sa nature, Spring WS n'est pas du tout adapté pour déployer des services existants sous forme de services Web.

Par contre, lors de la création d'un nouveau projet dès le départ orienté SOA, Spring WS devient très intéressant.

Pourquoi ?

D'un point de vue technique, Spring WS possède de nombreux avantages.

Tout d'abord, étant basé sur Spring dès le départ, il bénéficie de toutes les fonctionnalités inhérentes à celui-ci : injection de dépendances, AOP, intégration avec Spring Security (anciennement Acegi Security) ...

Par extension, Spring WS récupère l'expertise Spring des développeurs.

A coté de cela, Spring WS permet au développeur d'accéder aux messages sous différentes formes :

- API XML Standard : SAX, dom4j, JDom, StAx
- Objet Java sérialisé via JAXB, Castor, XMLBeans ou encore XStream

Au niveau interopérabilité, Spring WS supporte WS-Security et permet ainsi d'utiliser des moyens standards pour l'authentification/autorisation.

Evolution

Récemment, SpringSource a annoncé que la prochaine version majeure de Spring WS était en préparation et qu'elle apporterait son lot de nouveautés :

- Support de Spring 2.5 : les endpoint ne devront plus être déclarés automatiquement, mais pourront être configurés via l'annotation `@Endpoint`
- Support natif de Java 6
- Nouveau transport : JMS et Email en plus de HTTP
- Gestion des intercepteurs au niveau client
- Un namespace Spring dédié à Spring WS
- Un support de WS-Adressing

II - Cas pratique

Afin de montrer la facilité de Spring WS, nous allons créer un service simple. Celui-ci va simuler une demande de traduction de texte.

Les messages en entrée se composeront donc d'une langue d'origine, d'une langue de destination ainsi que le texte original. La réponse à cette demande contiendra le nom de l'auteur de la traduction, ainsi que le texte traduit.

Déploiement du service

Création du projet

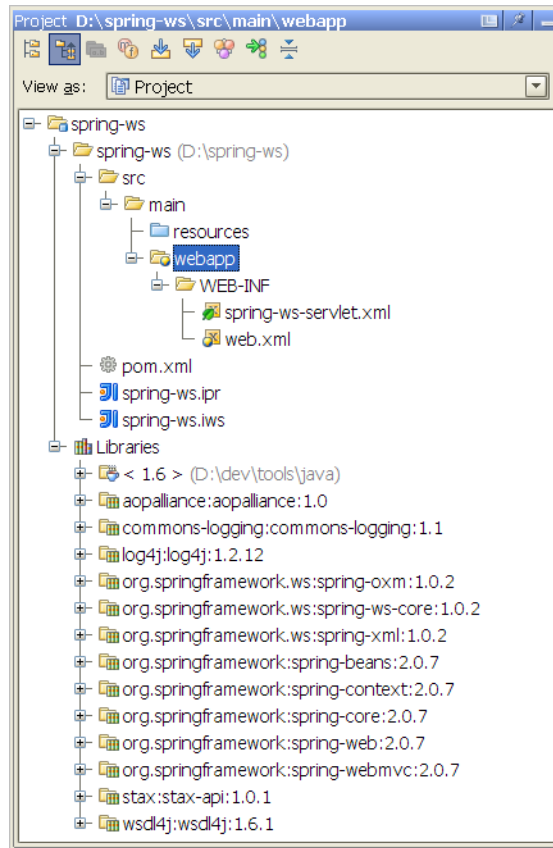
La facilité de Spring WS commence déjà à l'étape de la création du projet, en effet, afin de gagner du temps, un archétype Maven 2 permet la création d'un squelette du projet.

Pas besoin donc de chercher les dépendances nécessaires, cette responsabilité étant déléguée à Maven 2.

Dès lors, la création du projet se fait simplement par la commande suivant :

```
mvn archetype:create -DarchetypeGroupId=org.springframework.ws
-DarchetypeArtifactId=spring-ws-archetype \
-DarchetypeVersion=1.0.2 \
-DgroupId=<le groupe désiré> \
-DartifactId=<le nom du module>
```

Le projet ainsi créé, contient un fichier de configuration vide d'une servlet Spring MVC, ainsi que les dépendances de bases nécessaires au projet.

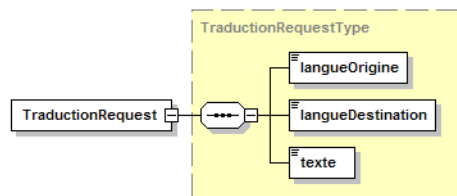


Arborescence du projet

Création du contrat

Une fois le projet créé, il est nécessaire de définir le contrat des données transférées, et ce, sous la forme d'un schéma XML.

La requête comprendra la langue d'origine du message, la langue vers laquelle une traduction est demandée ainsi qu'évidemment le texte à traduire :

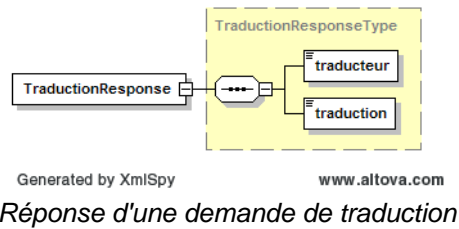


Generated by XmlSpy

www.altova.com

Requête de traduction

La réponse possèdera en plus du texte traduit, le nom du traducteur qui a réalisé la traduction



Réponse d'une demande de traduction

Voici le schéma XSD définissant ces messages :

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:traduction="http://www.hikage.be/schema/traduction"
  targetNamespace="http://www.hikage.be/schema/traduction" elementFormDefault="qualified">
  <xs:element name="TraductionRequest" type="traduction:TraductionRequestType"/>
  <xs:element name="TraductionResponse" type="traduction:TraductionResponseType"/>
  <xs:complexType name="TraductionRequestType">
    <xs:sequence>
      <xs:element name="langueOrigine" type="xs:string"/>
      <xs:element name="langueDestination" type="xs:string"/>
      <xs:element name="texte" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="TraductionResponseType">
    <xs:sequence>
      <xs:element name="auteur" type="xs:string"/>
      <xs:element name="texte" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

La nomenclature des éléments est très importante, le fait que ce soit **TraductionRequest** et **TraductionResponse** est en effet obligatoire pour permettre à Spring WS de faire la relation entre les deux au niveau des messages SOAP.


Implémentation de l'Endpoint

Une fois que les messages sont définis, l'étape suivante est de réaliser un endpoint, c'est à dire une classe qui va interpréter les messages en entrée, appeler le service de traduction proprement dit et renvoyer la réponse sous la forme définie précédemment.

Pour ce faire, Spring fournit différentes classes abstraites de base, afin de récupérer sous différentes formes le message. En effet, le message étant de l'XML, les moyens sont nombreux pour l'interpréter : SAX, JDOM, DOM4J, StAx, JAXB, Castor, ...

Dans cet exemple, nous allons utiliser un endpoint de type JDOM. Pour cela, c'est la classe **AbstractJDomPayloadEndpoint** qu'il faut étendre.

Il est intéressant de remarquer le **Payload** dans la classe. C'est à dire que l'élément JDOM qui sera récupéré ne contiendra que le contenu du message SOAP, c'est à dire ici, un TraductionRequest.

 *Si jamais il avait été utile de récupérer le message SOAP, et en particulier les Headers, il est possible de créer un MessageEndpoint, qui aura le message complet.*

Ajout de dépendances

Le projet crée par Maven au départ ne contient que les dépendances de base, mais pas les API XML proprement dites, simplement pour ne pas toutes les prendre par défaut si elles ne sont pas utilisées.

Il est donc nécessaire de les ajouter dans le POM :

```
<dependency>
  <groupId>jdom</groupId>
  <artifactId>jdom</artifactId>
  <version>1.0</version>
</dependency>
<dependency>
  <groupId>jaxen</groupId>
  <artifactId>jaxen</artifactId>
  <version>1.1</version>
</dependency>
```

Création de l'endpoint

Une fois ces dépendances créées, il est désormais possible d'implémenter l'endpoint proprement en étendant la classe qui va bien :

```
public class TraductionEndpoint extends AbstractJDomPayloadEndpoint {
    protected Element invokeInternal(Element request) throws Exception {
        Namespace namespace = Namespace.getNamespace("traduction",
            "http://www.hikage.be/schema/traduction");

        // Création des requête XPath pour récupérer les informations
        XPath langueOrigineExpression =
        XPath.newInstance("//traduction:TraductionRequest/traduction:langueOrigine");
        langueOrigineExpression.addNamespace(namespace);
        XPath langueDestinationExpression =
        XPath.newInstance("//traduction:TraductionRequest/traduction:langueDestination");
        langueDestinationExpression.addNamespace(namespace);
        XPath texteExpression =
        XPath.newInstance("//traduction:TraductionRequest/traduction:texte");
        texteExpression.addNamespace(namespace);

        // Récupération des informations à partir de la requête
        String langueOrigine = langueOrigineExpression.valueOf(request);
        String langueDestination = langueDestinationExpression.valueOf(request);
        String texteOriginal = texteExpression.valueOf(request);

        // Appel au service pour la traduction
        Traduction traduction = traductionService.traduitTexte(langueOrigine, langueDestination,
            texteOriginal);

        // Création de la réponse
        Element root = new Element("TraductionResponse", namespace);
        Element auteur = new Element("auteur", namespace);
        auteur.setText(traduction.getAuteur());

        Element texteTraduit = new Element("texte", namespace);
        texteTraduit.setText(traduction.getTexte());

        root.addContent(auteur);
        root.addContent(texteTraduit);

        return root;
    }
}
```

```
}  
}
```

Une fois l'endpoint créé, il reste à le déclarer dans le fichier de configuration de Spring :

```
<bean id="traductionEndpoint" class="be.hikage.spring.ws.endpoint.TraductionEndpoint">  
  <property name="traductionService" ref="traductionService"/>  
</bean>
```

Routage des messages vers le Endpoint

Une fois l'endpoint créé et configuré, il faut encore que les requêtes de traduction soient *routées* vers celui-ci.

Pour cela, Spring WS fournit une classe, qui va réaliser un mapping entre le type XML du message d'entrée et l'endpoint qui le gère.

Dans notre cas, cette configuration serait :

```
<bean class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">  
  <property name="mappings">  
    <props>  
      <prop  
key="{http://www.hikage.be/schema/traduction}TraductionRequest">traductionEndpoint</prop>  
      </props>  
    </property>  
    <property name="interceptors">  
      <list>  
        <bean  
class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor">  
          </list>  
      </property>  
</bean>
```

En clair, lorsque le corps d'un message SOAP contiendra un élément **TraductionRequest**, du namespace **http://www.hikage.be/schema/traduction**, celui-ci sera envoyé vers l'endpoint **traductionEndpoint**.

On remarque que la classe qui s'occupe du routage permet de configurer des intercepteurs. Un intercepteur est un objet qui a la possibilité d'agir sur un message AVANT qu'il ne soit interprété par un endpoint, mais aussi d'agir sur la réponse créée par celui-ci.

Dans l'exemple ci-dessus, il n'y a aucune action spécifique, cela ne fait qu'écrire dans les fichiers de logs les messages d'entrées et de sorties.

Mais il sera montré plus loin dans cet article l'utilité des intercepteurs, notamment dans la validation et la sécurité.

Publication du WSDL

A ce stade, le service web est fonctionnel, c'est à dire que si les bons messages sont envoyés à la bonne adresse, ils seront traités correctement.

Cependant, les clients ne disposent pas encore du WSDL, et donc il est quasi impossible d'y accéder correctement.

Il est donc nécessaire de mettre à disposition le WSDL, et pour cela c'est via la classe **DynamicWsd11Definition** que cela est possible :

```
<bean id="traduction" class="org.springframework.ws.wsdl.wsdl11.DynamicWsd11Definition">
  <property name="builder">
    <bean
      class="org.springframework.ws.wsdl.wsdl11.builder.XsdBasedSoap11Wsd14jDefinitionBuilder">
      <property name="schema" value="/WEB-INF/traduction.xsd"/>
      <property name="portTypeName" value="traduction"/>
      <property name="locationUri" value="http://localhost:9090/traductionService/">
    </bean>
  </property>
</bean>
```

Détaillons un peu cette configuration. L'identifiant **traduction** spécifie que le WSDL généré sera accessible en ajoutant **traduction.wSDL** à l'url de déploiement du service.

La propriété **schema** spécifie le chemin vers le schéma XML sur lequel Spring WS doit se baser pour générer le WSDL.

La propriété **portTypeName** spécifie le nom du portType dans le document WSDL.

La propriété **locationUri** spécifie l'url à laquelle sera accessible le service. Cette propriété ne vérifie pas que l'URL correspond réellement à la configuration réelle. Il faut donc s'assurer de donner la bonne URL.

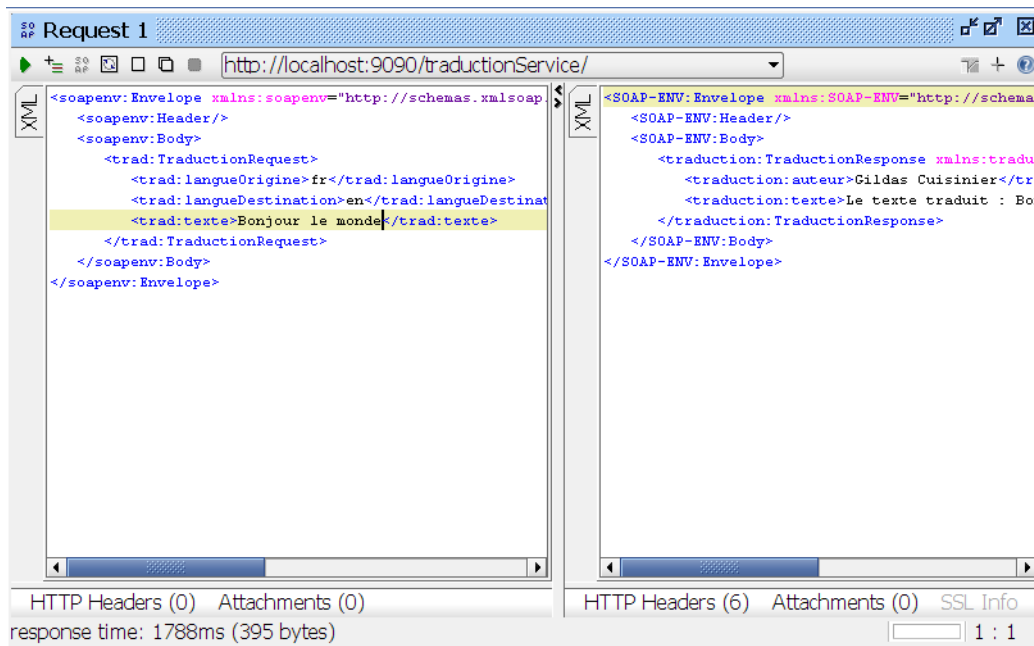
Test du service avec SoapUI

Le service Web nécessite de déployer le projet sous forme d'un WAR dans un serveur Tomcat ou autres.

Dans le projet fourni, il suffit de lancer la commande suivante pour tester :

```
mvn jetty:run
```

Une fois cela fait, il est possible de tester le service via **SoapUI** :



Création d'un client

Pour le côté client, Spring WS fournit une classe utilitaire : **WebServiceTemplate**.

Son utilisation est assez simple :

```

WebServiceTemplate webServiceTemplate = new WebServiceTemplate();

// Chargement d'une requête stockée dans un fichier
Resource resource = new ClassPathResource("be/hikage/spring/ws/client/traductionRequest.xml");

// Création d'un StreamSource, nécessaire pour l'utilisation de WebServiceTemplate
StreamSource source = new StreamSource(resource.getInputStream());

// Création d'un StreamDestination, ici redirigé vers System.out
StreamResult result = new StreamResult(System.out);

// Appel au service
webServiceTemplate.sendSourceAndReceiveToResult("http://localhost:9090/traductionService/", source,
result);
    
```

Dans cet exemple, la source de la requête est stockée dans un fichier, mais en pratique, elle devrait être créée dynamiquement à partir d'informations récoltées via une interface graphique.

Pareillement, ici le résultat est envoyé vers la sortie standard, mais devrait être récupéré pour être traité.

Mais l'exemple est assez explicite et montre les points importants : il n'y a pas de relation direct entre le WSDL et le client.

Et ici, il serait tout à fait possible d'envoyer n'importe quoi vers le service.

La partie cliente avec Spring WS est donc un peu plus complexe à mettre en oeuvre, mais il est de toute manière possible de créer un client avec n'importe quel autre API.

Cependant, il serait possible d'utiliser au niveau client des objets JAXB (ou autre marshaling géré par Spring WS). `WebServiceTemplate` possède des méthodes :

```
public Object marshalSendAndReceive(String uri, final Object requestPayload)
```

L'API Client de Spring WS permet aussi d'utiliser des `WebServiceMessageCallback`. Cela permet d'effectuer des traitements sur les messages avant leurs envois vers le service.

Cela sera démontré un peu plus loin lors de la gestion de la sécurité.

Améliorations du services

Validation

Par défaut, Spring WS ne valide pas les messages, ni en entrée, ni en sortie. C'est à dire que si un message `TraductionRequest` contient autre chose que les langues d'origine et de destination et le message à traduire, cela sera tout de même envoyé vers l'endpoint, au risque d'effectuer un mauvais traitement.

Afin de gérer cela, un intercepteur est proposé : **`PayloadValidatingInterceptor`**. Sa configuration est complètement gérée dans le contexte Spring :

```
<bean id="traductionValidatingInterceptor"
class="org.springframework.ws.soap.server.endpoint.interceptor.PayloadValidatingInterceptor">
  <property name="schema" value="/WEB-INF/traduction.xsd"/>
  <property name="validateRequest" value="true"/>
  <property name="validateResponse" value="true"/>
</bean>
```

Il prend en paramètre le schéma XSD utilisé pour la validation, et permet la validation indépendamment sur les messages en entrée et en sortie.

Cela permettrait ainsi d'être flexible sur les messages d'entrée, mais d'être rigide en ce qui concerne les réponses (qui doivent être comprises correctement par les clients, et donc doivent absolument être valides).

Comme tout intercepteur, il est ensuite nécessaire de l'ajouter en tant que tel :

```
<bean class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
  <property name="mappings">
    <props>
      <prop
key="{http://www.hikage.be/schema/traduction}TraductionRequest">traductionEndpoint</prop>
    </props>
  </property>
  <property name="interceptors">
    <list>
      <bean
class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor">
        <!-- Ajout de l'intercepteur de validation -->
        <ref bean="traductionValidatingInterceptor"/>
      </bean>
    </list>
  </property>
</bean>
```

Dès lors, si un client tente d'envoyer une requête mal formée, il se verra recevoir une faute SOAP:

```
<faultcode>SOAP-ENV:Client</faultcode>
  <faultstring xml:lang="en">Validation error</faultstring>
  <detail>
    <spring-ws:ValidationError xmlns:spring-ws="http://springframework.org/spring-ws">
      cvc-complex-type.2.4.a: Invalid content was found starting with element
      'trad:MauvaiseBalise'.
      One of '{"http://www.hikage.be/schema/traduction":langueOrigine}' is expected.
    </spring-ws:ValidationError>
  </detail>
```

WS-Security

Un autre exemple où les intercepteurs sont intéressants est la gestion de WS-Security.

Mais avant de mettre en oeuvre cela, il est nécessaire d'ajouter une dépendance au projet POM :

```
<!--
  Nécessaire pour l'utilisation de WS-Security
-->
<dependency>
  <groupId>org.springframework.ws</groupId>
  <artifactId>spring-ws-security</artifactId>
  <version>1.0.2</version>
</dependency>
```

Une fois cela fait, il est possible de configurer un intercepteur :

```
<bean id="wsSecurityInterceptor"
class="org.springframework.ws.soap.security.xwss.XwsSecurityInterceptor">
  <property name="policyConfiguration" value="classpath:/wss-server-config.xml"/>
  <property name="callbackHandlers">
    <list>
      <bean id="passwordValidationHandler"
class="org.springframework.ws.soap.security.xwss.callback.SimplePasswordValidationCallbackHandler">
        <property name="users">
          <props>
            <prop key="hikage">password</prop>
            <prop key="cafe">babe</prop>
          </props>
        </property>
      </bean>
    </list>
  </property>
</bean>
```

La première chose à configurer est la politique de configuration, qui prend un fichier de configuration WS Security en paramètre.

Voici un exemple qui prend en compte l'authentification :

```
<xwss:SecurityConfiguration xmlns:xwss="http://java.sun.com/xml/ns/xwss/config">
  <xwss:RequireUsernameToken passwordDigestRequired="false" nonceRequired="false"/>
</xwss:SecurityConfiguration>
```

Il demande un token d'authentification, mais pas obligatoirement chiffré.

Ensuite, il faut configurer les handlers qui vont s'occuper de l'authentification, ou encore de vérifier les signatures ou déchiffrer les messages lorsque cela est configuré.

Dans l'exemple, un seul handler est configuré, qui va vérifier les couples utilisateurs/mot de passe (basé sur une liste d'utilisateurs spécifiée directement). Il en existe d'autres plus évolués qui utilisent Spring Security pour ces vérifications.

Il faut ensuite ajouter l'intercepteur dans la chaîne, AVANT la validation :

```
<bean class="org.springframework.ws.server.endpoint.mapping.PayloadRootQNameEndpointMapping">
  <property name="mappings">
    <props>
      <prop key="{http://www.hikage.be/schema/traduction}TraductionRequest"
        >traductionEndpoint
      </prop>
    </props>
  </property>
  <property name="interceptors">
    <list>
      <bean
class="org.springframework.ws.server.endpoint.interceptor.PayloadLoggingInterceptor"/>
      <!-- Il est nécessaire de mettre l'intercepteur de sécurité avant la validation -->
      <ref bean="wsSecurityInterceptor"/>
      <ref bean="traductionValidatingInterceptor"/>
    </list>
  </property>
</bean>
```

Ici, le fait de le mettre avant ou après la validation n'a pas d'incidence, mais si le chiffrement était activé, l'intercepteur de validation ne comprendrait pas le message chiffré. Une fois que celui-ci sera passé dans l'intercepteur WS-Security, il sera déchiffré et compréhensible par la suite de la chaîne d'interception.

Dès lors, tout appel au service sans utiliser les entêtes WS-Security donnera :

```
<SOAP-ENV:Fault>
  <faultcode>SOAP-ENV:Client</faultcode>
  <faultstring xml:lang="en">
com.sun.xml.wss.XWSSecurityException: Message does not conform to configured policy
[ AuthenticationTokenPolicy(S) ]: No Security Header found; nested exception is
com.sun.xml.wss.XWSSecurityException: com.sun.xml.wss.XWSSecurityException:
Message does not conform to configured policy [ AuthenticationTokenPolicy(S) ]: No
Security Header found
  </faultstring>
</SOAP-ENV:Fault>
```

III - Conclusion

Comme vous l'avez remarqué, Spring WS propose donc une manière complètement différente de publier des services Web. Dans la première version du projet, le seul transport disponible est le protocole HTTP, qui est tout de même le plus courant. Mais la version 1.5 (actuellement en développement) apportera deux nouveaux transports : SMTP et JMS. Par ailleurs, la fonctionnalité d'intercepteur sera mise à disposition dans l'API coté client, ainsi qu'un support intégré à WS-Security, ces fonctionnalités étant accessibles uniquement coté serveur actuellement.

Je ferai une mise à jour de cet article lors de la parution officielle de Spring WS 1.5, afin de vous présenter plus en détails ces fonctionnalités.

En attendant, si vous désirez avoir plus d'informations, le [site officiel](#) possède une bonne documentation.

Remerciements

Je tiens à remercier djo.mos de ses précieux conseils pour l'écriture de cet article, ainsi que Dut et olsimare pour leur aide dans la correction de celui-ci.

