Savaş Ali TOKMEN
Master 2 Génie Informatique
Option Systèmes et Logiciels Embarqués
UFR IMA, Université Joseph Fourier, Grenoble
Février – Mars 2007

Envoi de données d'une passerelle OSGi™ vers un serveur Java EE via service web et exemple d'interface web d'accès aux données

Résumé: Dans cet article je vais décrire comment j'ai implémenté le système d'envoi de données d'une passerelle OSGi™ vers un serveur Java EE via un service web. Je décrirai comment le service web a été crée, comment ce service a été implémenté du côté Java EE ainsi que du côté OSGi™.

But et limites de l'application

Cette application fait la chose suivante:

- Du côté OSGi™, elle fournit un service `PushService` qui a une méthode: `pushMeasurement`. Cette méthode:
  - Prend comme argument le nom et le type de l'appareil de mesure ainsi qu'un arbre XML contenant les données à pousser.
  - Pousse les données vers le serveur Java EE de façon atomique, et jeter une exception si cette action échoue. Donc:
    - Si la méthode a jeté une exception alors aucune donnée n'a été sauvegardée du côté Java EE et
    - Toutes les données ont été sauvegardées de façon permanente sinon.
- La méthode doit pousser les données vers le serveur Java EE en utilisant un service web.
- Du côté Java EE, le serveur du service web doit sauvegarder les données reçues sous forme d'EJBs entité.

En particulier, les deux aspects suivants ne seront pas traités:

- Le parseage du XML (dernier argument de `pushMeasurement`): les données ont déjà été parsées et transformées en un objet Java représentant l'arbre XML de façon standardisée. Le format de l'arbre sera imposé par l'équipe développant le bundle qui utilisera le PushService.
- L'exploitation des données du côté Java EE: l'application va seulement sauvegarder les données reçues en tant que des EJB entité et les faire persister dans une base de données.

Technologies utilisées

On utilisera les technologies suivantes:

- Imposées par le sujet:
  - JOnAS 4.8.4 comme serveur Java EE
  - Felix 0.8.0 comme environnement OSGi™

- Imposées par l'auteur:
  - Apache ANT 1.7.0 avec la librairie bcel (c'est bien pratique!)
  - EasyBeans 1.0 Milestone 5 (pour le support EJB3)

- Imposées par les collègues:
  - JDOM 1.0 pour la réception des données par le bundle PushService (la représentation de l'arbre XML).

- Imposées par les technologies citées auparavant:
  - L'environnement Java 1.5 Update 11. On ne peut pas utiliser la version 1.6, car la version courante de JOnAS a des problèmes (=ne fonctionne pas correctement) avec Java 1.6.
  - JOnAS 4.8.4 utilisant JAX-RPC 1.1 (AXIS), cette technologie sera utilisée pour le serveur et client du service web en question.

Le service web

Interface du serveur web

Le langage WSDL décrit l'interface du service web, donc les méthodes disponibles, les arguments que ces méthodes prennent et les types de retours. On n'a en outre aucun commentaire (ou les commentaires ne sont pas correctement exploitées par java2wsdl et wsdl2java tout comme par de divers autres outils), de plus les noms des paramètres peuvent être perdues.

Ceci implique que par exemple la fonction suivante est un très mauvais candidat:

```
void pushMeasurement( String deviceName, String deviceType, String XMLContent );
```

car java2wsdl + wsdl2java produira le résultat suivant (chez le client du service web):

```
void pushMeasurement( String in0, String in1, String in2 );
```

où on aura donc perdu la signification des paramètres en question.

On notera tout de même que les noms des fonctions ainsi que les noms des structures et de leurs composants sont sauvegardées (que ce soit dans le sens Java vers WSDL ou WSDL vers Java). Il faut donc créer deux structures: une pour l'appareil de mesure et une autre pour une mesure:

```
public class Measurement implements java.io.Serializable {
	public Date date;
	public String data;
}
```

```
public class MeasurementDevice implements java.io.Serializable {
	public String type;
	public String name;
	public String pullIPAddress;
}
```

Comme les types `String` et `Date` sont prédéfinis dans le langage WSDL, ceci produit un WSDL:

- Qui ne perd pas les informations significatives (donc les noms des composants des structures décrivant les appareils et les mesures)
- Qui peut être utilisé par de divers implémentations de services web (Java, C#, etc.)

Pour pousser les mesures, il faut cette fois aussi utiliser un format qui est supporté par WSDL de façon natif: on doit donc utiliser des tableaux, et pas des `Collection`s ou des `List`s.

```
public interface MeasurementListenerService extends Remote {
	public void pushMeasurement( MeasurementDevice device, Measurement[] datum)
	                             throws RemoteException;
}
```

Ceci crée un WSDL qui ne contient que des types standards et où les significations de chaque argument comme de chaque partie de chaque structure utilisée est transparente; le fichier WSDL comme les fichiers Java générés par wsdl2java à partir de ce fichier nous le montrent bien. La seule difficulté que l'on observera est que la `Date` est transformée en une `Calendar`.

Côté serveur du service web

Bean stateless

Le côté serveur du service web utilisera un EJB de traitement qui a les propriétés suivantes:

- Il doit être **stateless**.
- Il doit utiliser des **transactions**, pour respecter la contrainte d'atomicité.

Ces deux propriétés sont implémentées en EJB3 en utilisant des annotations, donc il n'y a pas de problèmes particuliers. La classe distante de l'EJB a été nommée `MeasurementListenerRemote` et son implémentation `MeasurementListenerBean`.

Liaison bean stateless – service web

La liaison entre le service web (implémenté en tant que servlet par JOnAS et JAX-RPC) et le bean stateless se fait en utilisant le service de Naming proposé dans JOnAS. L'implémentation à proprement parler du service web (la classe `MeasurementListenerImpl`) va donc trouver le `MeasurementListenerBean` via un `Context.lookup` et directement appeler la méthode `pushMeasurement` sur la référence obtenue.

Les deux étapes citées (création d'un EJB stateless et son accès à partir d'un servlet) étant des étapes “classiques” d'un développement Java EE, on ne les détaillera pas plus.

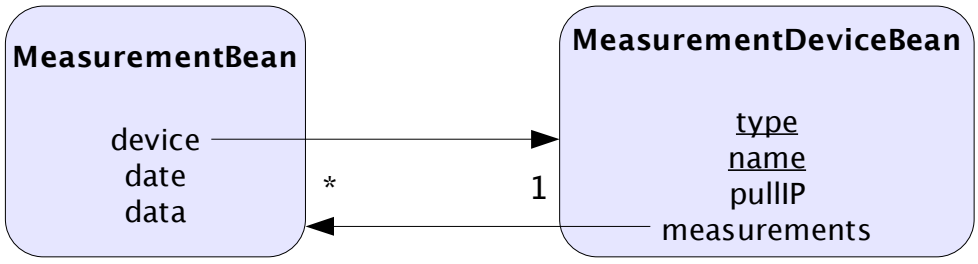
Une chose intéressante à remarquer est que le servlet est automatiquement généré par JOnAS et s'occupe de recevoir la requête SOAP, de le parser (transformer en objets Java) et d'appeler notre classe d'implémentation (POJO). Pour plus d'informations, merci de voir mon article à ce sujet: http://scholar.alishomepage.com/Master/JOnAS_OSGi/Article.pdf

Stockage: beans entité

Le côté serveur du service web utilisera un EJB de stockage qui a les propriétés suivantes:

- Il doit se rapprocher le plus possible des structures définies pour le service web
- L'insertion et la lecture de données doit se faire de façon relativement triviale
- On doit avoir le moins possible de duplications (respecter une forme BNF?)
- On doit être capable de retrouver rapidement la liste des appareils de mesure qui nous ont envoyés des données ainsi que la liste des mesures envoyées par un appareil donné

Pour respecter ces contraintes, le schéma suivant a été utilisé:



Les liaisons entre les classes `MeasurementBean` et `MeasurementDeviceBean` étant définis comme étant `ManyToOne` d'un côté et `OneToMany` de l'autre, on peut facilement insérer une mesure dans la liste des mesures d'un appareil: il suffit de dire que la mesure provient de l'appareil en question (donc de remplir la partie ManyToOne) et Java fera ce qu'il faut pour que la liste des mesures (partie OneToMany) de cet appareil soit mise à jour de façon automatique.

Côté client du service web

Interface d'appel

L'arbre XML est organisée de la façon suivante:

- Le nœud racine s'appelle “**Releves**”
- Sous ce nœud ce trouve zéro, un ou plusieurs nœud(s) “**Releve**”. Ces nœuds ont tous les deux éléments suivants:
  - Un nœud “**Date**”, qui contient comme valeur la date de la mesure. La date est représentée sous format “entier”, qui est le nombre de secondes écoulés depuis l'époque UNIX (1970).
  - Un nœud “**Donnees**” qui contient une valeur personnalisée en fonction du type d'appareil de mesure. Cette valeur sera lue et transmise directement en valeur, et peut donc contenir elle-même un arbre XML. Il se peut que le serveur Java EE fasse des traitement sur cette valeur, en fonction du type d'appareil de mesure.

Selon la spécification, le XML peut ne pas strictement suivre ce format. Dans ce cas:

- Si le nœud racine ne s'appelle pas “**Releves**”, alors une exception est jetée et le parseage ainsi que l'envoi est annulé.
- Si sous ce nœud se trouve un sous-nœud qui ne s'appelle pas “**Releve**”, ce sous-nœud est ignoré et un message de warning est affiché.
- Si le nœud “**Releve**” contient des sous-nœuds autre que “**Date**” et “**Donnees**”, alors ces sous-nœuds seront ignorés.
- Si le nœud “**Releve**” contient plusieurs sous-nœuds “**Date**” et/ou “**Donnees**”, alors seul le premier sous-nœud de chaque catégorie sera pris en compte.

L'implémentation du service `PushService` va donc premièrement transformer l'arbre XML en un tableau de `Measurement` en suivant cette spécification:

```
Element root = data.getRootElement();
if(root.getName().compareTo("Releves") == 0)
{
    MeasurementDevice device = new MeasurementDevice(
                                    InetAddress.getLocalHost().toString(),
                                    deviceType, deviceName);

    List children = root.getChildren();
    Iterator iterator = children.iterator();
    Measurement[] datum = new Measurement[children.size()];

    for(int i=0 ; i<datum.length ; i++) {
        Element child = (Element) iterator.next();
        if(child.getName().compareTo("Releve") == 0)
        {
            Calendar calendar = new GregorianCalendar();
            calendar.setTime( new
                              Date(Long.parseLong(child.getChild("Date").getText())));
            String data_ = child.getChild("Donnees").getValue();
            datum[i] = new Measurement(calendar, data_);
            System.out.println( "Will push data received at
                                 "+datum[i].getDate().getTime().toString());
        }
    }

}
```

Une fois l'arbre XML transformé en argument pour le service web, elle utilisera les classes générées par wsdl2java pour se connecter et envoyer les données vers le service web:

- Elle commencera par localiser le service web en utilisant la classe `Locator`, qui dans notre cas s'appelle `MeasurementListenerServiceServiceLocator`.
- En utilisant la localisation, elle obtiendra une connexion en utilisant la méthode `getMeasurementServicePort` de la classe `Locator`.
- À partir de ce moment, le service web peut être appelé. On va donc tout simplement appeler `pushMeasurement` sur la résultat de l'appel `getMeasurementServicePort`.

```
    // Connect to the web service and send data
    MeasurementListenerServiceServiceLocator locator =
              new MeasurementListenerServiceServiceLocator();
    MeasurementListenerService service = locator.getMeasurementServicePort();
    service.pushMeasurement(device, datum);
```

La localisation et l'appel se fait donc de façon totalement transparente et simple, en utilisant les classes générées par **wsdl2java**. Le seul handicap que ceci crée et que l'on est du coup obligé d'avoir les JAR AXIS associés (**~3 MO** au total).

Petits bémols

Dans le cadre de ce projet, le client du service web a été mis dans un bundle OSGi™ (nommé PushService). Ce bundle est un bundle qui reste en veille tant qu'aucun autre bundle n'a des choses à pousser et s'execute dans le même contexte (thread) que le bundle l'appelant.

Les bundles de notre projet sont complexes: ils utilisent des technologies différentes pour faire des choses similaires. Par exemple, le bundle PullService est basé sur le composant de service web XFire alors que le bundle PushService utilise AXIS. Par conséquent, ils utilisent certains classes communs de différentes façons (surtout les parseurs XML).

Ceci pose un problème: quand le PullService apelle (indirectement) le PushService, les parseurs XML dont le PullService a besoin sont chargés. Donc, le PushService jette des exceptions de cast de classes. Pour palier à ce problème, il faut obligatoirement changer le contexte d'execution du thread avant d'accéder aux éléments de l'arbre XML:

```
// This line avoids weird errors while using multiple XML parsers
Thread.currentThread().setContextClassLoader(this.getClass().getClassLoader());
```

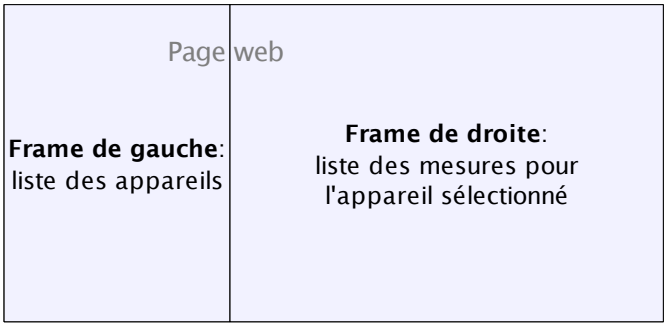
L'autre petit bémol est que nos bundles seront deployés et configurés à distance en utilisant JMX. Pour que ceci soit faisable, l'implémentation du bundle est dans une classe publique.

Accès aux données

Pour accéder aux données, un nouveau bean stateless nommé `MeasurementViewerBean` a été implémenté. Ce bean a les fonctionnalités suivantes:

- Lister tous les appareils de mesure. Comme ces appareils sont des `MeasurementDeviceBean`, cette liste peut aussi être utilisée pour accéder aux mesures.
- Obtenir l'appareil de mesure correspondant à un nom et type donné.
- Obtenir l'appareil de mesure en se basant sur son identificateur unique dans la base de données.
- Obtenir les *n* derniers mesures relevés par un appareil donné, ces mesures étant elles-mêmes ordonnées dans l'ordre décroissant suivant la date.
- Obtenir une mesure en se basant son identificateur unique dans la base de données.

La page qui liste les appareils de mesure et les mesures est divisée en deux frames HTML, comme le montre le schéma suivant:



Chaque frame pointe vers le servlet **MeasurementServlet**, qui sert aussi de point d'accès vers le bean stateless. Ce servlet peut être en trois modes:

- Mode “gauche”, où il accède à la liste des appareils de mesure et les donne au fichier JSP **left.jsp** pour le rendu.
- Mode “milieu” (ce qui correspond dans notre dessin à “droite), où il accède à la liste des mesures pour l'appareil sélectionné et les donne au fichier JSP **center.jsp** pour le rendu. Cette liste est accédée en utilisant le mode où seules les dernières mesures sont données.
- Mode “image”, où l'image correspondant à une mesure donnée (pour la webcam, donc) est redimensionnée et retournée.

Cette version initiale a tout de même été remplacée par une version plus avancée, où le traitement du frame de droite redirige vers un JSP plus adapté (vu que les trois modules retournent chacun des résultats très différents). Ces JSP ont été integrés dans de diverses applications:

- Le JSP rendant les températures construit des graphes.
- Le JSP rendant les coordonnés GPS les place sur Google Maps.
- Le JSP rendant les images de la webcam place une grande image pour le dernier instant et des petites images pour les instants précédents.

Pour que l'interaction soit plus vive, les JSP se rafraîchissent périodiquement.

Pour que le codage soit plus simple, la technologie JSTL a été intégrée dans les JSP. En outre, tous les moteurs de rendu de l'utilisent pas (c'est un choix!).

Conclusions

Si on énumère les bons côtés de cette implémentation:

- Le service web a été implémenté de façon compatible. Il fonctionne avec le client initial (passerelle OSGi™) mais aussi avec un client de test écrit en C#.
- Le service web ainsi que les EJBs dont il dépend fonctionnent correctement sous JOnAS 4.
- Les transactions sont atomiques.
- Le parseage côté OSGi™ fonctionne correctement et semble ne pas être trop gourmand.

Les spécifications ont donc été satisfaites.

D'un autre côté, il aurait peut-être été intéressant d'avoir des beans de stockage supplémentaires:

- Stocker les températures capturées par le thermomètre directement en flottant
- Stocker les positions capturées par le GPS directement dans une structure adéquate, plus facilement exploitable
- Stocker les images capturées par la webcam directement en format image, donc ne pas avoir à décoder du base64 dans le servlet à chaque fois.

En addition à ceux-ci, une technologie plus avancée (AJAX?) aurait pu être utilisée pour le dynamisme du côté de l'interface web (la version actuelle ne fait que rafraichir la page entière!).

C'est dommage de ne pas l'avoir fait, mais je n'avais pas eu le temps...