MINES NANTES

Cahier des charges

Projet d'option CXF

Auteurs :

Grégoire Henry-Séguin
Kévin LLopart
Raphaël Martignoni
Hao Zhang

Octobre 2013

Première version du cahier des charges avant validation par le tuteur du projet

Table des matières

1. Présentation du projet

1.1. Présentation du framework CXF

Apache CXF est un framework dédié au développement de services web en langage Java. Il résulte de la combinaison de deux projets open-source par des membres de l'Apache Software Foundation. Le framework inclut de nombreuses fonctionnalités, parmi lesquelles les API JAX-RS (REST) et JAX-WS (SOAP).

1.2. Introduction au problème

Deux propriétés sont absentes du Framework CXF.

La première concerne **le couplage entre les deux couches du framework**. Ce dernier est composé d'une couche objet et d'une couche service. Il serait intéressant que ces deux couches soient faiblement couplées, afin de permettre un déploiement d'un même modèle objet avec deux services différents. Un même modèle objet pourrait alors être utilisé avec les deux technologies REST et SOAP incluses dans CXF. A ce jour, il existe un couplage entre ces deux couches empêchant une telle utilisation du framework de façon simple (apparition de bugs).

La deuxième propriété est relative à l'interopérabilité, induite par le principe de substitution. L'interopérabilité pourrait permettre de communiquer un objet allant du client au serveur, qui soit un sous-type du type normalement attendu.

La partie suivante met en évidence l'absence de ces propriétés dans le framework CXF, de façon empirique.

1.3. Objectifs du projet

Le projet consiste à rendre ces deux propriétés effectives dans CXF.

2. Mise en évidence dans CXF

2.1. Observations

Dans une première phase de pré-projet, l'absence d'interopérabilité et de couplage fort a été mise en évidence dans CXF, à travers des exemples utilisant JAX-RS et JAX-WS. A noter que les exemples ne montrent pas une absence systématique des propriétés nommées. Au contraire, les bugs renvoyés par CXF dépendent à la fois de la version du framework et de la technologie web utilisée (SOAP ou RESTful). Nous relatons ici quelques bugs illustrant le problème décrit. L'ensemble des exemples est accessible sur github.[1]

2.2. Résultats

2.2.1. SOAP

Exemple avec substitution de l'interface.

Un client est initialement créé sous Eclipse à partir du code du serveur. Le web service présente initialement une méthode op(Eleve e). Cette dernière est modifiée après la génération du client. L'argument en entrée est remplacé par un sur-type d'Eleve: Personne. Notons qu'Eleve possède des champs supplémentaires, absents de la classe Personne. Le client fait ensuite appel à l'opération op, avec en argument une instance d'Eleve. On obtient l'erreur suivante :

Exception in thread "main" javax.xml.ws.soap.SOAPFaultException: Unmarshalling Error: unexpected element (URI : "", local : "promo"). Expected elements are <{}prenom>,<{}nom>

Cette erreur met en évidence un couplage fort entre la couche objet et la couche service de CXF.

2.2.2. RESTful

[1] Lien du projet : https://github.com/zerbino/cxfemn/
Pour les exemples détaillés, rapports>log[0..n].org

3. Améliorations à apporter

Le document de référence[2] utilisé pour ce projet d'option indique les nouvelles spécifications à ajouter à CXF pour que le principe de substitution soit respecté. Nous ne rappellerons pas ces nouvelles spécifications dans ce document, pour ne nous en tenir qu'à l'interprétation de ces résultats et aux conclusions quant à l'implémentation.

Interprétation des nouvelles spécifications

Dans le cas RESTful :

Lorsqu'un document docB de type GB est en entrée, le framework doit successivement appeler :

- La fonction de démarshallisation en lui passant un type B
- La fonction de conversation de B vers A
- La fonction de mashallisation en lui passant un type A pour récupérer un document doc_A de type G_A.

Dans le cas SOAP :

Pour un document doc_{FB} de type G_{FB}, le framework doit successivement appeler :

- La fonction de démarshallisation en passant un type F_B, produit par le fonction de génération sur B
- La fonction de projection, correspondant à un appel de getter de la classe commande de FB
- La fonction de conversation de B à A
- L'appel au constructeur de la classe commande F_A en passant A en argument, et un setter
- La fonction de marshallisation en passant le type F_A, et en récupérant un document doc_{FA}, avec un type G_{FA}.

Algorithme de lifting

Si l'on s'en tient à l'interprétation des spécifications, plusieurs contraintes apparaissent :

- Les types A et B doivent être connus du framework
- La conversion ne se fait pas de façon efficace car elle implique une traduction document-objet suivie d'une traduction objet-document.

Une solution plus efficace serait donc de procéder à une transformation de document à document. Avec JAXB, en utilisant XML, les documents diffèrent de part :

- Le nom du root tag
- Des éléments additionnels correspondant à des attributs supplémentaires

Ainsi, pour passer d'un document docB à un document docA, il suffit de renommer le root tag et de supprimer les éléments absents du deuxième document. Cette suppression correspond à un algorithme connu : le « tree inclusion problem »

---

[2] *The Substitution Principle in an Object-Oriented Framework for Web Services : From Failure to Success.* Collaborateurs : Diana Allam, Hervé Grall et Jean-Claude Royer, ASCOLA, Mines de Nantes - INRIA

Déploiement de l'implémentation

Les modifications à apporter au projet sont à inclure entre la marshallisation des données à l'émission par le client (resp. serveur) et la démarshallisation lors de la réception par le serveur (resp. client).

Une première solution serait d'apporter cette modification à l'intérieur même du framework, par modification de la fonction de marshallisation et de démarshallisation. Cette solution implique la modification du databinding de chaque framework ou la création complète d'un nouveau framework de databinding.

Ces deux solutions étant couteuses, la solution réside dans l'utilisation d'intercepteur. L'architecture de CXF permet en effet d'insérer un intercepteur dans la chaine d'intercepteur déjà existante. Celui – ci serait placé dans la chaine d'intercepteur entrante, à la réception.

La partie suivante décrit plus en détail ce qu'est un intercepteur et son utilisation.

4. Utilisation des intercepteurs

4.1. Qu'est ce qu'un intercepteur ?

Les intercepteurs sont utilisés pour intercepter les messages qui transitent entre le serveur et le client, dans un sens ou dans l'autre. L'intercepteur peut agir sur le message en le lisant, le transformant, lui modifiant son en-tête, le validant, etc.

Lors de l'invocation d'un serveur CXF par un client CXF, une **InterceptorChain** de sortie est générée pour client, et une **InterceptorChain** d'entrée est créée pour le serveur. A l'inverse, lorsque le serveur renvoie une réponse au client, une chaine de sortie est créée pour le serveur, et une d'entrée est fournie au client.

Les InterceptorChains sont divisées en phases, correspondant aux étapes de transit et traitement du message. Chaque phase contient plusieurs intercepteurs correspondant. Pour une chaine d'entrée, on aura par exemple les phases suivantes :

Phase	Functions
RECEIVE	Transport level processing
(PRE/USER/POST)_STREAM	Stream level processing/transformations
READ	This is where header reading typically occurs.
(PRE/USER/POST)_PROTOCOL	Protocol processing, such as JAX-WS SOAP handlers
UNMARSHAL	Unmarshalling of the request
(PRE/USER/POST)_LOGICAL	Processing of the umarshalled request
PRE_INVOKE	Pre invocation actions
INVOKE	Invocation of the service
POST_INVOKE	Invocation of the outgoing chain if there is one

4.2. Comment l'utiliser ?

4.2.1. Ecriture de l'intercepteur

Pour écrire un intercepteur, il faut étendre la classe AbstractPhaseInterceptor ou l'une de ses sous-classes. Il faut préciser dans le constructeur à quelle phase l'intercepteur sera appelé.

Exemple :

```
import java.io.IOException;

import org.apache.cxf.attachment.AttachmentDeserializer;
import org.apache.cxf.message.Message;
import org.apache.cxf.phase.AbstractPhaseInterceptor;
import org.apache.cxf.phase.Phase;

public class AttachmentInInterceptor extends AbstractPhaseInterceptor<Message> {
    public AttachmentInInterceptor() {
        super(Phase.RECEIVE);
    }

    public void handleMessage(Message message) {
        String contentType = (String) message.get(Message.CONTENT_TYPE);
        if (contentType != null && contentType.toLowerCase().indexOf("multipart/related") != -1) {
            AttachmentDeserializer ad = new AttachmentDeserializer(message);
            try {
                ad.initializeAttachments();
            } catch (IOException e) {
                throw new Fault(e);
            }
        }
    }

    public void handleFault(Message messageParam) {
    }
}
```

Ici l'intercepteur est appelé lors de la phase de réception.

4.2.2. Ajout de l'intercepteur

Pour ajouter l'intercepteur à un client ou un serveur, on l'ajoute à un InterceptorProvider.

L'interceptorProvider est une interface de ce type :

```
public interface InterceptorProvider {

    List<Interceptor> getInInterceptors();

    List<Interceptor> getOutInterceptors();

    List<Interceptor> getOutFaultInterceptors();

    List<Interceptor> getInFaultInterceptors();
}
```

Pour ajouter l'instance d'intercepteur à l'InterceptorProvider, on procède comme suit :

```
MyInterceptor interceptor = new MyInterceptor();
provider.getInInterceptors().add(interceptor);
```

Les InterceptorProvider sont de plusieurs types :

- Client
- Endpoint
- Service
- Bus
- Binding

La récupération de l'InterceptorProvider et l'ajout d'un intercepteur à celui-ci se fait comme suit :

Dans le cas du serveur :

```
import org.apache.cxf.endpoint.Server;
import org.apache.cxf.frontend.ServerFactoryBean;
...

MyInterceptor myInterceptor = new MyInterceptor();

Server server = serverFactoryBean.create();
server.getEndpoint().getInInterceptor().add(myInterceptor);
```

Dans le cas du client :

```
import org.apache.cxf.endpoint.Client;
import org.apache.cxf.frontend.ClientProxy;
...

MyInterceptor myInterceptor = new MyInterceptor();
FooService client = ... ; // created from ClientProxyFactoryBean or generated JAX-WS client

//You could also call clientProxyFactroyBean.getInInterceptor().add(myInterceptor) to add the interceptor

Client cxfClient = ClientProxy.getClient(client);
cxfClient.getInInterceptors().add(myInterceptor);

// then you can call the service
client.doSomething();
```

5. Planification du projet