SOA facile avec SCA

Christophe Demarey, Damien Fournier

SOA facile avec SCA

Comment simplifier le développement d'applications SOA tout en se donnant un cadre architectural ? SCA et notamment FraSCAti que nous utiliserons apportent des réponses à ces préoccupations.
http://frascati.ow2.org

Introduction

Ecrire des applications SOA, avec de nombreux services web, n'est pas toujours chose aisée. Notamment, la mise en œuvre de services web (WS, REST, etc.) demande du temps et surtout du code technique en plus de vos classes métiers. Que diriez-vous de n'écrire que le code métier et simplement spécifier dans un fichier XML les services que vous voulez exposer sur le web ? SCA rend ceci possible ! Mais ce n'est pas le seul avantage, SCA vous permet aussi de bénéficier d'un cadre architectural pour vos applications orientées services. Enfin, il permet de mixer des applicatifs utilisant des technologies différentes (bundle OSGi, Java, scripts, BPEL, etc.) et des protocoles de communication hétéroclites (SOAP, HTTP, JSON-RPC). Dans cet article, nous ne reviendrons pas sur les fondamentaux de SCA, présentés dans le numéro 110.

Présentation et installation de FraSCAti

Il existe plusieurs implémentations des spécifications SCA (OW2 FraSCAti, Apache Tuscany, IBM WebSphere, etc.). Nous utiliserons FraSCAti, une plate-forme open-source du consortium OW2. Elle ne supporte pas tous les langages et protocoles spécifiés pour SCA (focus sur les technologies Java) mais, en contrepartie, fournit plusieurs fonctionnalités avancées, telles que le support de composants SCA réflexifs (permettant le changement de configuration à chaud des assemblages), des protocoles d'accès originaux comme UPNP, JNA ou encore des outils qui nous aiderons à observer l'état et administrer les assemblages de composants pendant leur exécution. FraSCAti est téléchargeable sous la forme d'une archive zip sur le site du projet (http://frascati.ow2.org). Pour installer la plateforme, il suffit d'extraire les fichiers de l'archive.

SCA, cadre architectural

Notre exemple fil rouge, MyWeather, consiste en une orchestration de plusieurs services. Dans un premier temps, nous allons interroger un compte twitter pour récupérer la localisation d'une personne, ensuite nous interrogerons un service météo afin d'obtenir le temps pour ce lieu.

SCA propose un langage d'architecture décrit en XML qui permet de construire des applications par assemblage de composants, le composant représentant une fonctionnalité implémentée généralement par une classe. Un assemblage SCA est comparable à la notion de contexte d'application dans Spring, les beans étant assimilés à des composants SCA.
Il est possible de générer le descripteur d'architecture (fichier *.composite) en utilisant l'éditeur STP/SCA d'Eclipse, mais nous allons ici modéliser notre exemple directement en XML [Figure 1.]. Pour cet assemblage (ou composite) SCA nous devons réaliser deux composants :
- le composant Decoder qui traduit les messages XML renvoyés par le service météo,

- le composant Orchestration qui utilise les services twitter, météo et le composant Decoder. Il définit une propriété SCA utilisée pour configurer l'identifiant du compte twitter.

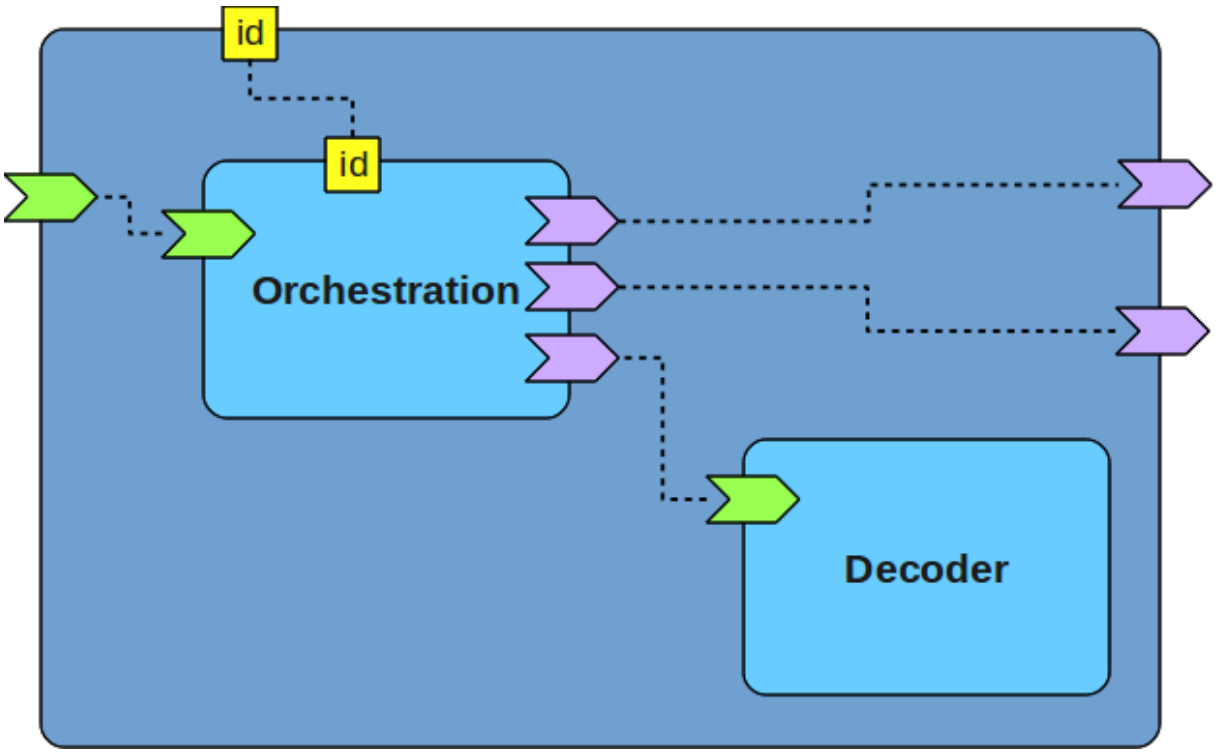

Figure 1: Implémentation du composant de décodage

SCA, vecteur d'interopérabilité et d'intégration technologique

SCA n'impose pas de langage particulier pour ses composants. Il est possible d'implémenter des composants avec le langage Java ou Scala, des scripts Ruby ou JavaScript, ou encore d'utiliser des composants Spring. Cette flexibilité se retrouve pour la création ou l'utilisation de services pouvant être réalisés par le biais des technologies SOAP, HTTP, RMI, JMS, etc. Les nombreux langages et protocoles d'accès supportés par les implémentations de SCA présentent cette technologie comme solution efficace pour faciliter l'intégration de composants logiciels existants.

Implémentation d'un composant SCA

Nous allons, dans un premier temps, nous intéresser à l'implémentation de notre composant de décodage qui sera utilisé pour traduire les réponses du service météo, encodées au format XML. Notons que ce composant ainsi que les objets JAXB sont nécessaires car le WSDL du service décrit le type de retour de ces méthodes comme une chaine de caractères et non le type réel (liste de Locations).
Le composant Decoder, développé en Java, fournit une méthode 'decode'. Nous commençons par définir une interface Java, nommée 'Decoder', qui représentera le contrat du service. Cette interface a la signature suivante : Locations decode(String message);
On nommera son implémentation DecoderImpl:

```
 public Locations decode(String message) {
  InputStream is = new ByteArrayInputStream(message.getBytes());
  try {
   JAXBContext context = JAXBContext.newInstance(Locations.class);
   Unmarshaller unmarshaller = context.createUnmarshaller();
   return (Locations) unmarshaller.unmarshal(is);
  } catch (JAXBException e) {
   System.err.println("Unable to decode server message.");
   return null;
 } }
```

Cette méthode permet de décoder (via JAXB) le message XML renvoyé par le service météo et donnant la liste des villes pour lesquelles le service météo est disponible. Cette liste est utilisée par le composant d'orchestration pour connaitre l'ensemble des villes où les informations météo sont accessibles.

Notons qu'un composant SCA ne peut exposer un service que s'il fournit une interface décrivant ses méthodes (aussi appelée contrat). Ici, le contrat est une interface Java, mais SCA permet aussi de définir les contrats via des descripteurs WSDL.

SOA simplifiée

Grâce à SCA, nous allons très facilement utiliser ou réaliser des services distants sans nous soucier des détails techniques propres aux protocoles d'accès. Nous illustrerons l'intégration et l'exposition de services avec SCA via la création d'un composant d'orchestration qui proposera la mise à jour des informations météorologiques pour des utilisateurs du service twitter.

Pour obtenir les informations relatives à un compte utilisateur, twitter propose une API conforme au principe d'architecture REST (REpresentational State Transfer). Le service de météo est lui, accessible via le protocole SOAP. Nous allons accéder à ces services de nature différente via les connecteurs REST et Web Service fournis par la plateforme FraSCAti. L'utilisation d'un connecteur avec SCA est non intrusive, elle n'impacte pas l'implémentation même du composant. C'est FraSCAti qui se chargera de la création des stubs/squelettes en charge des connexions distantes.

Accès à un service REST : twitter

Nous passons à l'intégration des services twitter et météo afin de les rendre utilisables par notre orchestration. Pour rappel, la technologie SCA propose l'intégration des services via de nombreux protocoles d'accès dont l'utilisation est spécifiée via le descripteur d'architecture. Par exemple, pour accéder aux méthodes exposées par twitter nous sélectionnerons le binding REST en ajoutant, dans la description de notre composite SCA, la balise XML suivante:

```
 <frascati:binding.rest uri="http://twitter.com"/>
```

L'attribut URI permet d'indiquer l'adresse de la ressource web à accéder via REST, dans notre cas "http://twitter.com". Pour permettre à notre composant, implémenté en Java, d'accéder aux informations du profil utilisateur, Nous devons écrire l'interface qui reflètera le service utilisé.

```
import javax.ws.rs.*;
```

```
public interface Twitter {
  @GET
  @Path("/users/show/{id}.xml")
  User getUser(@PathParam("id") String id);
}
```

Nous utiliserons la méthode "getUser" qui permet de récupérer les informations contenues dans le profil utilisateur. Cette méthode prend en paramètre l'identifiant de l'utilisateur. Dans le cas d'un service REST, nous devons utiliser les annotations de l'API JAX-RS afin de préciser le chemin d'accès, la commande HTTP, et les paramètres utilisés pour accéder à la ressource. L'appel à la méthode "getUser" renvoie un objet de type User reflétant l'architecture de la ressource REST reçue. La classe User comporte les annotations de l'API JAXB afin de réaliser la correspondance entre les tags XML et les champs de l'objet. Le code java pour la classe "User" est donné ci-dessous :

```
import javax.xml.bind.annotation.*;

@XmlRootElement
public class User {
   public String id;
   public String name;
   public String screen_name;
   public String location;
   ...
}
```

La valeur de l'attribut "location" de l'objet "User" contient les informations de localisation renseignées dans le compte twitter. Nous pouvons passer à l'intégration du service météo.

Accès à un Web Service (SOAP) météo

Le service météo que nous utilisons est accessible via le protocole SOAP, il est publié à l'adresse "http://www.webservicex.net/globalweather.asmx". Ce service web présente deux opérations : "GetCitiesByCountry" qui permet de lister les villes où les informations sur la météo sont disponibles et "GetWeather" pour connaitre la météo d'une ville. Pour utiliser ce service nous allons faire appel au connecteur SOAP de la plateforme FraSCAti. Dans notre descripteur d'assemblage, nous décrivons l'accès à ce service via une référence comportant un binding web service. Le service météo est alors rendu accessible pour les composants SCA de notre assemblage, et donc accessible pour le composant d'orchestration. Dans le descripteur d'assemblage SCA, cette liaison se traduit par la balise suivante :

```
<binding.ws
   wsdli:wsdlLocation="http://www.webservicex.net/globalweather.asmx?wsdl"

wsdlElement="http://www.webservicex.net#wsdl.port(GlobalWeather/GlobalWeather
Soap)" />
```

La présence des attributs wsdlLocation et wsdlElement permettent à la plateforme de retrouver le descripteur de l'interface WSDL et le port d'accès au service web. Pour rendre le service météo utilisable par le composant d'orchestration, nous devons générer l'interface Java reflétant ce service à partir de son descripteur WSDL. Pour réaliser cette opération FraSCAti propose la commande wsdl2java :

% frascati wsdl2java -u http://www.webservicex.net/globalweather.asmx?wsdl -o src/generated
Elle permet d'obtenir l'interface "GlobalWeatherSoap" dans le dossier des sources Java. L'interface obtenue est décorée avec les annotations de l'API JAX-WS, indiquant le nom des opérations, paramètres et messages proposés par le service web. C'est via cette interface que l'on pourra interroger le service météo.

Réalisation de l'orchestration
Nous avons réalisé le composant de décodage des messages, puis écrit/généré les interfaces qui permettent d'interagir avec les services externes : twitter et météo. Nous pouvons, à présent, nous occuper de l'implantation du composant d'orchestration qui se chargera d'agréger les informations récupérées des services externes et de décoder les informations de localisation. Nous commençons par créer la classe "Orchestration" suivante :

```
public class Orchestration {

  @Reference protected Twitter twitter;

  @Reference protected Decoder decoder;

  @Reference protected GlobalWeatherSoap weather;

  @Property protected String userId;
   ...

}
```

Dans notre implémentation, nous utilisons deux annotations spécifiques à SCA. L'annotation "@Reference" précise les attributs de la classe permettant d'utiliser d'autres services, internes (ici le decoder) ou distants (twitter et météo dans notre cas). L'injection d'une référence vers un autre service ou un autre composant est réalisée par la plateforme SCA. La seconde annotation "@Property" définit une propriété configurable du composant, la valeur de cette propriété sera définie dans le descripteur d'assemblage SCA.
Il ne nous reste qu'a implémenter la méthode d'orchestration que nous nommerons "getWeatherForUser()". Nous allons d'abord appeler le service twitter et vérifier que notre utilisateur a renseigné la localisation dans son profil.

```
public String getWeatherForUser() {
   User user = twitter.getUser(userId);
   if (user.location.equals("")) {
     System.err.println("The user " + userId + " did not publish his location");
     return "N/A";
   } ...
```

Nous supposons que l'attribut de localisation comporte le nom de ville et du pays séparés par une virgule, que nous récupérons dans les variables "cityName" et "countryName"

```
   ...
   String[] locations = user.location.split("[\\s]*,[\\s]*", 2);
   String cityName = locations[0];
   String countryName = locations[1];
```

```
   System.out.println("User '" + userId + "' is living in " + cityName
      + " (" + countryName + ")"); ...
```
On récupère la liste des villes proposées par le service météo en utilisant la référence correspondante : "weather", puis on appelle le composant de décodage. On obtient alors une instance d'objet "Locations".
```
   String cities = weather.getCitiesByCountry(countryName);
   Locations l = decoder.decode(cities);
   ...
```
Enfin, on compare la liste des villes avec la localisation de l'utilisateur. S'il y a correspondance, on appelle de nouveau le service météo pour obtenir les conditions météo détaillées:
```
   String result = "";
   if (l != null) {
     boolean done = false;
     for (Location loc : l.locations) {
       if (loc.city.value.toLowerCase().contains(cityName.toLowerCase())) {
         result += "Current weather in " + loc.city.value + ":\n";
         result += weather.getWeather(loc.city.value, countryName) + "\n";
         done = true;
     } }
     if (!done) {
       System.err.println("Unable to find '" + cityName
          + "' in available cities of the weather service");
   } }
   return result;
```

Descripteur d'assemblage

Nous avons implémenté les différents composants et interfaces nécessaires pour réaliser notre orchestration. Nous pouvons maintenant décrire l'architecture de notre application SCA. Le descripteur d'architecture appelé "composite" permet de construire notre application en définissant les composants, leurs liaisons, les propriétés de configuration ou encore les connecteurs utilisés.

L'écriture d'un composite commence avec l'élément XML "composite" qui mentionne obligatoirement un attribut "name". Notez également la définition des espaces de nommages via l'attribut "xmlns", qui seront ensuite utilisés dans la description:
```
<composite name="twitter-weather"
 xmlns="http://www.osoa.org/xmlns/sca/1.0"
 xmlns:wsdli="http://www.w3.org/2004/08/wsdl-instance"
 xmlns:frascati="http://frascati.ow2.org/xmlns/sca/1.1">
....
```
On définit ensuite les composants SCA. Dans un premier temps, nous ajoutons le composant responsable du décodage des messages XML. Les lignes suivantes permettent de nommer le composant et de lui associer la classe d'implémentation Java écrite précédemment.
```
 <component name="decoder">
   <implementation.java
class="org.ow2.frascati.examples.twitterweather.lib.DecoderImpl" />
 </component>
```

Nous ajoutons le second composant, responsable de l'orchestration. Outre la classe d'implémentation, nous décrivons les références vers les services nommés twitter et weather, ainsi que vers le composant de décodage. Nous définissons également la propriété "userId" qui utilisera la valeur définie dans la propriété nommée "userId" du composite (réutilisation de propriétés).

```
 <component name="orchestration">
   <implementation.java
class="org.ow2.frascati.examples.twitterweather.lib.Orchestration"/>
   <reference name="twitter"/>
   <reference name="weather"/>
   <reference name="decoder" target="decoder"/>
   <property name="userId" source="$userId"/>
 </component>
```

Finalement, on décrit les références vers les services en spécifiant les connecteurs utilisés et on définit une valeur pour la propriété userId au niveau du composite.

```
  <reference name="twitter" promote="orchestration/twitter">
    <frascati:binding.rest uri="http://twitter.com"/>
  </reference>

  <reference name="weather" promote="orchestration/weather">
    <binding.ws
wsdli:wsdlLocation="http://www.webservicex.net/globalweather.asmx?wsdl"
      wsdlElement="http://www.webservicex.net#wsdl.port(GlobalWeather
/GlobalWeatherSoap)" />
  </reference>

  <property name="userId">userid</property>
</composite>
```

Créer un service web

Pour exposer notre service TwitterWeather et le rendre accessible via SOAP, nous allons ajouter une interface le décrivant. Nous nommerons cette interface "TwitterWeather". Notez l'utilisation de l'annotation @Service pour indiquer que l'interface Java sera utilisée comme contrat pour le service SCA. Voici l'interface qu'implémentera alors la classe Orchestration.

```
@Service
public interface TwitterWeather {
  String getWeatherForUser();
}
```

On ajoute ensuite la définition du service au niveau du composite.

```
<service name="tw" promote="orchestration/TwitterWeather">
   <binding.ws uri="http://localhost:8080/TwitterWeather"/>
</service>
```

Le service "tw" sera ainsi accessible soit via un appel à parti d'un client SOAP, soit localement.

Il ne nous reste plus qu'à compiler et exécuter notre intégration de service implémentée avec la technologie SCA. Pour créer le jar de l'application, nous appelons FraSCAti avec la commande de compilation:

```
% frascati compile src myWeather
```

L'exécution est ensuite réalisée grâce à la commande "run". On précisera les noms du service (option –s) et de la méthode (option –m) à appeler ainsi que le chemin vers le jar de l'application.

```
% frascati run myWeather -libpath myWeather.jar -s tw -m getWeatherForUser
```

Conclusion

Nous avons illustré que SCA permet de construire très rapidement des architectures orientées services pouvant utiliser différentes technologies de communication. Nous avons réalisé une petite application, en quelques lignes de code, réalisant une orchestration entre un service SOAP et une ressource REST. Toutefois SCA et FraSCAti ne se limitent pas à cet usage. Dans un prochain article nous irons plus loin avec FraSCAti. Nous utiliserons ses capacités réflexives et son interface d'administration puissante pour montrer, avec un script ou en quelques clics, qu'il est possible de modifier le comportement d'une application SCA, de modifier sa configuration ou d'exposer les services d'une application encore plus rapidement.