**Examen de Web Services**

**Ing2, Orientation SIE**

**Janvier 2013 (2h00)**

**Conditions d'examen :**

* Deux feuilles recto verso manuscrites ;
* Pas d'Internet, pas de réseau, ni aucune communication ;
* Le devoir se fait sur papier.

**Questions :**

1) JAX-RS est-il un framework ou une spécification ?

JAX-RS est une spécification, l'une des implémentations (framework) de cette spécification est Jersey.

2) Rappelez les 3 phases du paradigme SOA ?

Chercher, publier, consommer (cf slide 9 du premier cours)

3) Qu'est ce que la découvrabilité d'un service ? Comment s'appelle le fournisseur associé ?

La découvrabilité d'un service résulte de la mise à disposition du fichier WSDL ou WADL par le fournisseur du Web Services, sur un annuaire prévu à cet effet.

4) Peut-on écrire le code d'un client consommant un Web Services en ne connaissant ni le code, ni le langage de programmation de ce dernier ? Décrire les différentes étapes à suivre en cas de réponse positive.

Il est tout à fait possible d'écrire le code d'un client consommant un Web Services en ne connaissant ni le code ni le langage utilisé. C'est d'ailleurs l'un des points les plus importants concernant les Web Services.

Il y a plusieurs façons de procéder. L'utilisation du fichier WSDL ou du fichier WADL permet de connaitre et d'utiliser un Web Services de façon automatisée, nous avons pu le remarquer à l'aide de SOAP-UI, logiciel capable d'extraire les informations de ce type de document, pour en extraire les points d'entrée/retour disponible pour un fichier WSDL/WADL donnée, et donc, d'un Web Services.

D'une façon générale, nous avons besoin de connaitre les paramètres et les points d'entrée, et de même pour la sortie, une fois cela fait, nous n'avons pas besoin de plus pour utiliser ce Web Services.

Les possibilités "cherche l'annuaire" ou autre, sont un avantage, mais pas nécessairement obligatoire.

5) Peut-on mélanger JAX-WS et JAX-RS dans le même système ?

Il est tout à fait possible de mélanger JAX-WS et JAX-RS dans un même service. Beaucoup le font (notamment Amazon, et il y a peu Google).

Attention : nous parlons bien ici de JAX-WS et JAX-RS, c'est à dire les spécifications. Donc tous les frameworks découlant de ces deux normes, seront donc compatible.

Attention : toutefois, SOAP ne définit pas HTTP PUT et HTTP DELETE par défaut, mélanger les deux sur la même fonction n'est donc normalement pas à faire, même si dans les faits, là encore, c'est techniquement réalisable.

**Exercice 1 :**

 Gravatar est un Web Services qui permet de lier une adresse email à une photo de son choix. Il suffit ensuite, sur un site qui supporte l'API Gravatar (par exemple tous les sites sous Wordpress), de stipuler la même adresse email lors de l'enregistrement pour automatiquement récupérer la bonne image. La gestion de l'avatar est donc déportée sur le Web Services.

Ce service, lancé par l'équipe de Wordpress, est particulièrement populaire du à sa très grande simplicité d'utilisation.

 On s'occupera ici, de recréer, en REST, une version similaire de ce programme, non plus avec les avatars, mais cette fois-ci avec les données de l'utilisateur (login, mot de passe, nom, prénom, ...).

On considère que l'on travaille sous Jersey (comme vu en cours), donc sous Java.

* Créez une classe Java nommé "GrUser" ;
* Dans cette classe, intégrez quatre fonctions qui permettent de manipuler les données de l'utilisateur : une pour ajouter, une pour modifier, une pour consulter et enfin une pour supprimer l'utilisateur ;
* On ne s'occupe pas de la sécurité du système ;
* On ne s'occupe pas de l'intérieur de chaque fonction ;
* On suppose que nous disposons d'un objet "User" qui permet de manipuler les utilisateurs.

1) Proposez, a l'aide des annotations vu en cours/TD, une interface claire pour que notre Web Services puisse être accessible de l'extérieur (via plusieurs URLs, de la façon la plus simple possible).

Attention : beaucoup d'entre vous confondent JAX-WS et JAX-RS, hors ici il était demandé JAX-RS (Utilisation de Jersey dans l'énoncé).

Attention : on ne demandait nullement la classe user ou autre (puisque nous disposions de cette classe, c'est qu'elle existe déjà...), juste un modèle vide mais avec annotations de GrUser...

@Path("GrUserTest")

public class GrUser {

 @POST

 @Produces(MediaType.APPLICATION\_JSON)

 public User create() {}

 @GET

 @Path("{id : \\d+}")

 @Produces(MediaType.APPLICATION\_JSON)

 public User read(@PathParam("id") int id) {}

 @PUT

 @Path("{id : \\d+}")

 @Consumes(MediaType.APPLICATION\_JSON)

 @Produces(MediaType.APPLICATION\_JSON)

 public User update(@PathParam("id") int id, User user) {}

 @DELETE
 @Path("{id : \\d+}")

 public void delete(@PathParam("id") int id) {}

}

Rappels :

1) @Path définit un morceaux (ou la totalité) du chemin à suivre, ici le @Path du début de la classe va définir une base que tous les autres @Path vont utiliser à l'intérieur de cette classe (cf question 2)

2) @Produces définit la façon dont la réponse est traitée. En effet, nous renvoyons un objet java, celui-ci ne peut rester tel quel puisqu'il doit être compréhensible par tous, donc nous appliquons une transformation JSON dessus (proche d'une transformation XML). Ce qui nous permet d'être compatible avec tout langage capable de lire du JSON (soit a peu près tous les langages existants).

3) @Consumes réalise la même chose que @Produces, mais dans le sens inverse : ce qui arrive à la fonction, est traité et transformé en Objet Java. Nous envoyons donc un JSON comme suit :

{

 "login" : "bonjour",

 "email" : "test@eisti.eu"

}

Et nous obtenons dans les paramètres de la fonction (voir @PUT) un objet User directement. La conversion étant assurée par Jersey lui même.

4) @Path("{id : \\d+}") Nous utilisons ici une expression régulière pour limiter id aux seuls nombres :

\\d+ :

 \d signifie "un nombre"

 + signifie "1 ou plusieurs"

Le double "\" est du à l'escape réalisé par Java, \\ done un seul \ en sortie.

Cette petite amélioration n'était pas exigée.

2) Proposez l'url finale d'accès à la modification (on utilisera comme base localhost, comme port 8080, et le nom du projet s'appelle GrUser)

Par rapport à ce que nous avons fait, nous avons donc comme URI finale :

http://localhost:8080/GrUserTest/{id : \\d+}

Ou id est un nombre qui change à chaque requête, le JSON est quant à lui embarqué dans le corps de la requête, il n'a donc pas à apparaitre ici.

ex : http://localhost:8080/GrUserTest/15

3) Proposez deux façons d'envoyer des données pour deux fonctions (on suppose donc ici deux fonctions de modifications, avec deux façons de stipuler les paramètres). Présentez les URLs correspondantes et le contenu associé (avec comme paramètres minimum : le login, le mot de passe, le nom de famille et le prénom).

Changer @PUT par @GET ou @POST comme beaucoup l'ont fait, n'était pas acceptable ici puisque l'on casse dans le paradigme REST : GET pour obtenir, POST pour ajouter, PUT pour modifier, DELETE pour supprimer une ressource. Ainsi changer @PUT par @GET ou @POST n'est clairement pas acceptable puisque l'on demande une modification (donc deux fois @PUT).

Il existe de nombreuses façons de le faire, je décris ici deux méthodes, souvent employées dans vos copies :

Nous avons vu à la question 1 l'utilisation de @Consumes, nous allons voir ici @PathParam, et @QueryParam.

Il existait aussi @FormParam par exemple.

@PUT

@Path("{id : \\d+}/{login: \\w+}/{password: \\w+}/{lastname: \\w+}/firstname: \\w+}")

@Produces(MediaType.APPLICATION\_JSON)

public User update(@PathParam("id") int id, @PathParam("password") String pass, @PathParam("login") String log, @PathParam("lastname") String lastname, @PathParam("firstname") String firstname) {}

Ici on se rend clairement compte que @Consumes offre un gain de temps non négligeable. De plus, cette version refuse l'utilisation de "/" dans les variables, puisque ce caractère est utilisé comme délimiteur.

De même, j'ai volontairement inverser password et login dans les paramètres de la fonction, pour montrer qu'il n'y a pas d'incidence à le faire, la relation entre la variable et le paramètre dans @Path se faisant à partir du texte donné en paramètre dans @PathParam.

De la même façon, une variable qui n'a pas le même nom que celui dans @PathParam, ne pose pas de soucis.

Par contre, si @PathParam n'a pas une chaine contenue dans @Path, alors, cette variable ne sera jamais correctement remplie...

L'url finale d'accès sera :

http://localhost:8080/GrUserTest/15/monlogin/monpassword/monnom/monprenom

Note : \\w+ se comporte comme \\d+, mais cette fois ci pour le texte : il est autorisé n'importe quel caractère alphanumérique ici.

La deuxième façon est similaire dans l'idée :

@PUT

@PathParam("{id: \\d+}")

@Produces(MediaType.APPLICATION\_JSON)

public User update(@PathParam("id") int id, @QueryParam("password") String pass, @QueryParam("login") String login, @QueryParam("firstname") String firstname, @QueryParam("lastname") String last) {}

Ici tout ce qui était valable pour la première version (le comportement dans la correspondance des paramètres est sur le même principe). Cependant, si l'on regarde @Path, on ne trouve aucune référence à login, password, ou autre.

Cela vient du fait qu'un paramètre de requête, n'est vu de la même façon que l'uri. Dans l'étape 1, nous passions les paramètres dans l'uri directement, ici nous passons les paramètres en query param (GET PARAM) :

http://localhost:8080/GrUserTest/15/?login=monlogin&password=monpassword&lastname=monnom&firstname=monprenom

L'utilisation de ? et & vous est certainement familière (beaucoup de site web utilise cela), et dans ce cas ci, les paramètres existent bien, mais ils sont interprétés comme paramètre et non comme partie intégrante de la requête, ainsi ils peuvent être récupérés, mais pour autant n'apparaissent pas dans @Path.

On peut alors utiliser @DefaultParam(quelquechose) pour palier au fait que le paramètre existe ou non :

public User update(@PathParam("id") int id, @DefaultParam("monlogin") @PathParam("login") String login, ...)

Cette annotation aura pour qualité de mettre une valeur par défaut si le paramètre n'est pas précisé.

Note : on préfère toujours si possible le passage via @Consumes, ou @PathParam plutôt que @QueryParam, pour deux raisons : c'est plus lisible, c'est mieux référencé par les moteurs comme Google. La syntaxe étant plus simple (côté utilisateur du service), elle est donc mieux comprises de tous.

**Bottom Up**

A l’inverse, cette approche prône une phase de conception ascendante. Une analyse de l’existant (**cartographie applicative**) permet de déterminer les fonctions existantes du SI. Ainsi cartographiées, il est possible d’identifier les fonctions du SI qui sont éligibles au rang de service. Une fois ces fonctions mises en mode service, elles sont exploitables au sein de services à forte valeur ajoutée et / ou de processus métiers.

Cette approche peut séduire par plusieurs aspects :

* Elle contraint à réaliser une cartographie du SI qui, si elle est tenue à jour et publiée, facilitera la réutilisation de services.
* Le coup du ticket d’entrée peut paraître moins élevé que pour une démarche Top Down.

Elle présente cependant des travers rédhibitoires :

* Elle bloque le pilotage de la SOA par les besoins métiers.
* Elle ne favorise pas la mise en place d’un effort transverse : elle ne permet pas de sortir de la « culture projet ». En effet, les services émergeants de cette approche restent très fortement couplés à leur application d’origine.
* Elle rend très difficile la justification de l’investissement auprès du métier, qui n’en verra les bénéfices qu’une fois les services auront été rendus exploitables au sein de processus métiers.

Enfin, cette approche se limite à une mise en mode service de fonctions existantes sur le SI. C’est présupposer que cet existant suffit à réaliser le métier de l’entreprise. Quel est dans ce cas la pertinence de la mise en place d’une SOA pour aligner le SI sur le métier ?