



silverpeas

API DE SERVICES WEB REST DANS SILVERPEAS

Version :	V1.0
Auteurs :	Miguel Moquillon
Approbation :	
Date modification :	11/03/2011

Table des matières

1 Introduction.....	3
2 Jersey.....	4
3 L'API REST de Silverpeas.....	9

1 INTRODUCTION

Dans le cadre du développement de widgets AJAX sur les commentaires, le framework Jersey a été introduit dans Silverpeas avec pour objectif d'exposer, sous forme de services Web de type REST (de préférence RESTful), les ressources gérées par Silverpeas (publications, images, commentaires, etc.) ; ceci permet l'accès à ces ressources en AJAX à partir de l'IHM.

La force de développer une telle couche service est de pouvoir satisfaire deux besoins de façon homogène :

- étendre les appels AJAX dans la couche IHM de Silverpeas afin d'améliorer l'interactivité ; il s'agit donc de remplacer les widgets en JSP par leurs équivalents en javascript (sans présager de la technologie utilisée), quitte à les encapsuler par des tags JSTL.
- remplacer les appels aux EJB et les widgets graphiques dans la taglib website par respectivement des appels HTTP et des widgets en javascript.

Mais aussi d'être en mesure de monter plus facilement en charge.

Ainsi, avec une couche de service Web selon l'approche architecturale REST, il s'agit de permettre aussi bien à des sites Web extérieurs qu'à la couche graphique de Silverpeas de pouvoir interroger les objets métiers ; il améliore la séparation des responsabilités de la couche Web d'avec celle métier de Silverpeas.

L'écriture d'un premier service Web REST pour publier sur le Web les commentaires a permis de mettre en place d'une part les frameworks qui facilitent le développement et les tests de tels services, et d'autre part de dessiner une première version d'une API pour la réalisation et les tests de services Web dans Silverpeas.

2 JERSEY

[Jersey](#) est l'implémentation de référence de la JSR-311, ou JAX-RS, qui introduit sur la plate-forme Java une façon normalisée et simplifiée (et non simpliste) d'écrire des services Web RESTful.

Avant de développer un service Web REST, il est vivement recommandé de s'informer de ce qu'est [REST et de sa philosophie sous-jacente](#).

2.1 L'APPROCHE REST

[REST ou REpresentational State Transfer](#) a été développé par Roy T. Fielding dans le cadre de sa thèse sur [les styles architecturaux des applications réseau](#).

REST est une approche architecturale qui prend à contre pied celle issue des premières applications distribuées et qui est encore en vigueur. L'approche classique est de concevoir un système distribué comme un réseau d'appels distants avec une nette tendance d'abstraire, dans la mesure du possible, les appels aux opérations de façon à imiter les appels natifs d'un logiciel monolithique ; pour ce faire, la nature réseau des appels se doit être cachée. Ceci a abouti à des architectures de type [RPC](#) qui ont été unifiées pour la plupart avec l'architecture [CORBA](#). La finalité de cette approche s'est retrouvée dans le [SOA](#) avec [SOAP](#) qui, pour schématiser, est, en quelque part, un rejeton de CORBA pour le Web. Si le SOA n'impose pas [HTTP](#) comme protocole réseau de transport, il n'en est pas moins qu'il s'est rapidement développé avec lui comme média de communication. En effet, HTTP est un protocole de transport dans la couche [OSI](#) « application », ce qui fait qu'il peut être utilisé sur n'importe quel couche réseau de transport (TCP/IP, WDP, ...) ; ainsi, avec HTTP, il est plus facile de migrer les frameworks SOA sur différents types de réseau.

Toutefois, pour des applications Web, l'approche SOA ajoute une surcharge et une complexité qui est loin d'être contrebalancée par ses qualités. De plus, la nature même du protocole HTTP contrevient à l'approche SOA (qui reste principalement stateful et qui, par nature, nécessite un mode connecté).

C'est ici que l'approche REST intervient. L'objectif de ce dernier est de proposer un style d'architecture pour les éléments (logiciels ou non) distribués sur *un réseau hypermédia* (le terme hypermédia est important parce qu'il sous-tend la communication d'informations numérique). Ainsi REST rompt avec l'approche traditionnelle en mettant l'accent non plus sur les opérations publiées par les services, mais sur les ressources hypermédia exposées par les services. Les opérations sur ces ressources sont alors celles inhérentes au protocole de communication utilisé et sous-jacent à l'architecture. Ainsi, pour une application Web, le protocole HTTP avec ses opérations (appelées méthodes) POST, PUT, DELETE, GET, HEAD, OPTIONS sont utilisées avec leur propre sémantique. Donc, ce ne sont plus les données relatives aux opérations invoquées qui sont transférées d'un point à l'autre, mais les ressources hypermédia, ou plus exactement, comme son nom l'indique, leur état sous un format négocié entre le consommateur et le fournisseur de la ressource ; un service Web selon l'approche REST est un service stateless.

Sur le Web, les ressources, selon une architecture REST, sont uniquement identifiées par une URI et c'est par celle-ci qu'elles sont accédées. Une ressource peut être aussi bien un média (texte structuré, image, vidéo, ...) qu'un état donné d'un processus par exemple. Les méthodes du protocole HTTP sont utilisées pour interagir avec les ressources publiées :

- POST pour la demande de création d'une ressource à partir de l'état initial de celui-ci transmise dans la requête sous un format convenu (en général en XML ou en JSON),

- GET pour la demande de l'état courant de la ressource identifiée par l'URI à laquelle la requête a été envoyée. L'état est retourné dans un format convenu entre le client et le service,
- PUT pour la mise à jour de la ressource (et dans certaines architectures la création de celle-ci si elle n'existe pas) avec l'état transmis dans la requête dans un format convenu,
- DELETE pour la suppression de la ressource identifiée par l'URI à laquelle a été envoyé la requête,
- HEAD pour vérifier l'existence d'une ressource donnée et obtenir des méta-informations sur celle-ci,
- OPTIONS pour obtenir des informations sur les moyens de communication possible avec la ressource identifiée par l'URI à laquelle la requête a été envoyé. Ce genre de requête peut être utilisé pour connaître les différents formats supportés par le service pour encoder l'état de la ressource.

En général, ce sont des ressources dédiées, sorte de conteneur, qui répondent aux requêtes POST. Les requêtes PUT devrait être envoyées à la ressource concernée par la mise à jour, mais dans certaines architectures ce peut être des ressources dédiées, comme pour les requêtes POST, qui peuvent répondre aux mise à jour.

Le format dans lequel l'état des ressources est transféré entre le client et le service est indiqué par le paramètre d'entête HTTP *Content-Type*. Lors de la création d'une nouvelle ressource, l'URI de celle-ci doit être retournée dans le paramètre d'entête HTTP *Location*. Lorsqu'une ressource référence d'autres ressources exposées, ces références sont représentées dans l'état transmis entre le client et le serveur sous forme d'URI afin de permettre la navigation d'une ressource à l'autre selon le mécanisme d'hyperliens.

Les code HTTP sont aussi utilisées pour informer le statut de la requête :

- 200 si la requête a été correctement traitée,
- 201 si la demande de création a aboutie,
- 400 si la requête a été mal formulée ; par exemple, le format dans lequel est transféré l'état de la ressource n'est pas comprise par le service,
- 401 si l'accès à la ressource nécessite une authentification et la requête ne s'inscrit pas dans une session ouverte,
- 403 si le client n'a pas les droits suffisants pour accéder à la ressource demandée,
- 405 si la ressource cible de la requête ne supporte pas l'opération HTTP,
- 409 si, par exemple, la ressource à créer existe déjà ou la ressource à mettre à jour ne correspond pas avec celle dont l'état est transmis dans la requête,
- 415 si le format dans lequel est transmis la ressource n'est pas supporté par le service,

- 424 ou 503 pour signaler que le traitement de la requête a échoué pour une raison donnée,
- 500 si une erreur imprévue est survenue lors du traitement de la requête,
- ...

2.2 DÉVELOPPER UN SERVICE WEB AVEC JERSEY

Écrire un Web service REST est facile avec Jersey. Cette section prend pour exemple le service Web de gestion des commentaires `CommentResource`.

Dans Jersey, n'importe quel POJO peut devenir un service Web à partir du moment qu'il est annoté avec l'annotation `@Path` qui indique la racine de l'URI à partir duquel est accessible le service :

```
@Path("comments/{componentId}/{contentId}")
public class CommentResource
```

Le chemin indiqué dans l'annotation `@Path` est relative à l'URI de base à laquelle la servlet Jersey est déployée et à l'écoute des requêtes vers les services Web REST.

Par défaut, tout objet de `CommentResource` sera associé à une requête et donc aura une durée de vie en conséquence. Pour qu'un service Web ait une durée de vie au delà de la requête, il faut alors l'annoter avec `@Singleton`.

Les parties comprises entre les accolades ouvrantes et fermantes dans le chemin définissent les parties variables. Si la durée de vie de l'objet s'inscrit dans une requête, il est alors possible de récupérer leurs valeurs via les attributs d'instance avec l'annotation `@PathParam` et le nom de la partie variable :

```
@PathParam("componentId") private String componentId;
@PathParam("contentId") private String contentId;
```

Dans le cas d'un service Web avec une durée de vie au delà de la requête, la valeur des parties variables ne pourront être récupérées que par les arguments de l'opération qui prend en charge la requête HTTP et ceci avec la même annotation `@PathParam`.

Les requêtes à destination du service Web sont traitées par ses méthodes. Elles sont automatiquement identifiables par une annotation qui indique le type de la requête HTTP pris en charge :

- `@GET` pour les requêtes HTTP de type GET,
- `@POST` pour les requêtes HTTP de type POST,
- `@PUT` pour les requêtes HTTP de type PUT,
- `@DELETE` pour les requêtes HTTP de type DELETE,
- `@HEAD` pour les requêtes HTTP de type HEAD,
- et `@OPTIONS` pour les requêtes HTTP de type OPTIONS.

D'autres paramètres peuvent être fournis sous forme d'annotation comme le sous-chemin dans l'URI qui est traité par la méthode, le format dans lequel l'état de la ressource est attendu ou envoyé, etc. :

```

@GET
@Path("/{commentId}")
@Produces(MediaType.APPLICATION_JSON)
public CommentEntity getComment(@PathParam("commentId") String onCommentId)

```

Ici ci, la méthode `getComment` sera invoquée à la réception d'une requête HTTP de type GET à l'URI `comments/{componentId}/{contentId}/{commentId}`. La valeur de la partie variable (`commentId`) sera récupérée par le paramètre `onCommentId` avec l'annotation `@PathParam`.

L'annotation `@Produces` indique les formats dans lesquels l'état de la ressource, ici le commentaire, peuvent être renvoyés. Dans l'exemple, seul le format JSON est supporté et le client doit donc bien indiquer le type MIME de ce format dans le champs d'entête HTTP *Accept*. L'objet renvoyé par la méthode, de type `CommentEntity`, sera sérialisé en JSON par Jersey avec par défaut le parseur JAXB.

Le pendant à l'annotation `@Produces` est `@Consumes` qui indique les formats sous lesquels le service Web peut récupérer l'état des ressources ; le client doit indiquer avec le champs d'entête *Content-Type* le type MIME du format dans lequel l'état d'une ressource est transmise :

```

@POST
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response saveNewComment(final CommentEntity commentToSave)

```

Ou dans le cas d'un service Web supportant aussi le format XML :

```

@POST
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
@Consumes({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response saveNewComment(final CommentEntity commentToSave)

```

Par défaut, dans une requête POST, le paramètre non annoté recevra le résultat de la désérialisation de l'état de la ressource embarquée dans le corps de la requête HTTP.

Cette méthode renvoie une réponse Web, ici représentée par l'objet `Response` qui est construit de la façon suivante dans la méthode `saveNewComment` :

```

Response.created(commentURI).entity(asWebEntity(savedComment,
identifiedBy(commentURI))).build();

```

La réponse est construite avec une code HTTP 201 (*CREATED*) et dans laquelle l'état de l'objet créé est embarqué.

Il est aussi possible de récupérer des informations de contexte lors d'une invocation du service Web comme par exemple l'URI à laquelle le service Web a été accédé ou la requête HTTP sous-jacente :

```

@Context
private UriInfo uriInfo;
@Context
private HttpServletRequest httpRequest;

```

De même, les entêtes HTTP peuvent aussi être obtenues via l'annotation `@HeaderParam` et il est possible de préciser des valeurs par défaut aux paramètres ou aux variables à récupérer avec l'annotation `@DefaultValue` :

```
@DefaultValue("") @HeaderParam(HTTP_AUTHORIZATION)
private String credentials;
@DefaultValue("") @HeaderParam(HTTP_SESSIONKEY)
private String sessionKey;
```

3 L'API REST DE SILVERPEAS

Dans l'objectif de faciliter l'écriture de services Web REST dans Silverpeas et surtout de les tester, un début d'API a été écrite. Le service sur les commentaires fait usage de cette API.

3.1 PUBLIER UN SERVICE WEB REST DANS SILVERPEAS

L'API s'appuie à la fois sur les frameworks Jersey et Spring :

- Jersey prend en charge les requêtes vers les services Web et les réponses de ceux-ci aux clients. Il gère le mécanisme d'identification du service Web accédé et l'exécution de ses méthodes. Les services Web dans Silverpeas seront déployées automatiquement sous le chemin `services/` du contexte Web de l'application.
- Spring prend en charge le cycle de vie des services Web et fournit à Jersey l'accès à ceux-ci. Il suffit alors juste d'indiquer à Spring où trouver nos services Web via un fichier de configuration de contexte Spring.

Pour ce faire, une servlet provenant de Jersey et s'appuyant sur le framework Spring est défini dans le descripteur Web de Silverpeas sous le nom de *REST Container*. Cette servlet est à l'écoute de toute requête vers le chemin `/services/` de l'URI. Le reste du chemin permet d'identifier le service Web interrogé et doit être définie par l'annotation `@Path` à chaque service Web.

Pour que Spring puisse prendre en charge le cycle de vie des services Web et donc qu'il puisse fournir à Jersey les services REST disponibles, il est nécessaire aussi d'annoter ceux-ci avec `@Service` :

```
@Service
@Scope("request")
@Path("comments/{componentId}/{contentId}")
public class CommentResource
```

Et d'indiquer, via un fichier de configuration de contexte Spring, avec le tag XML `component-scan`, le package Java à partir duquel le ou les services sont définis :

```

<?xml version="1.0" encoding="UTF-8"?>
<!--

Copyright (C) 2000 - 2009 Silverpeas

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.

As a special exception to the terms and conditions of version 3.0 of
the GPL, you may redistribute this Program in connection with Free/Libre
Open Source Software ("FLOSS") applications as described in Silverpeas's
FLOSS exception. You should have received a copy of the text describing
the FLOSS exception, and it is also available here:
"http://repository.silverpeas.com/legal/licensing"

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

-->

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:ctx="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

  <ctx:annotation-config/>
  <ctx:component-scan base-package="com.silverpeas.comment"/>

  ...

```

Dans Spring les classes d'objets sont gérés par défaut comme des singletons tandis que dans Jersey les services sont instanciées par requêtes. Il est alors nécessaire d'indiquer à Spring d'associer le cycle de vie des services Web avec les requêtes entrantes grâce à l'annotation Spring @Scope valorisée à request (attention, il existe aussi une annotation @Scope dans Jersey).

Dans l'exemple ci-dessus, le service CommentResource est publié sous le chemin de l'URI /services/comments/{componentId}/{contentId} où componentId et contentId sont les parties variables du chemin.

3.2 ÉCRIRE UN SERVICE WEB REST

3.2.1 La classe des services Web REST RESTWebService

Tout service Web dans Silverpeas doit étendre la classe de base RESTWebService. Cette classe définit un service Web dans Silverpeas et fournit des méthodes permettant l'authentification, l'autorisation d'accès à la ressource interrogée, et l'accès à des informations comme celles relatives à l'URI de la requête, les préférences de l'utilisateur, etc.

```
@Service
@Scope("request")
@Path("comments/{componentId}/{contentId}")
public class CommentResource extends RESTWebService {
```

Si le service Web nécessite que l'accès soit authentifié et autorisé, l'exécution de chacune de ses méthodes doivent alors débiter par la vérification des privilèges de l'utilisateur avec l'appel à la méthode héritée `checkUserPriviledges()` :

```
@Service
@Scope("request")
@Path("comments/{componentId}/{contentId}")
public class CommentResource extends RESTWebService {

    @GET
    @Path("/{commentId}")
    @Produces(MediaType.APPLICATION_JSON)
    public CommentEntity getComment(@PathParam("commentId") String onComment) {
        checkUserPriviledges();
        ...
    }
    ...
}
```

Cette méthode requière que la méthode abstraite `getComponentId()` soit implémentée par le service Web.

3.2.2 Les erreurs et les exceptions

Si le traitement de la requête aboutie à une erreur, il faut la convertir en une erreur HTTP qui correspond au problème rencontré :

- code d'erreur 400 (BAD REQUEST) si l'état de la ressource transmis dans la requête n'est pas conforme à ce qui est attendu, suite à un import personnalisé ou à des attributs non correctement valorisés,
- code d'erreur 404 (NOT FOUND) si la ressource demandée ou à mettre à jour n'existe pas,
- code d'erreur 409 (CONFLICT) si une ressource à créer existe déjà ou si la ressource à mettre à jour ne correspond pas à l'état transmis dans la requête,
- code d'erreur 424 (METHOD, FAILURE, code réservé pour WebDAV) ou code d'erreur 503 (SERVICE UNAVAILABLE) si une exception métier est survenue (erreur exceptionnelle, donc qui ne devrait pas survenir en temps normal),
- code d'erreur 500 (INTERNAL SERVER ERROR) si une exception technique survient lors du traitement de la requête.

Les erreurs relatives à un accès non authentifié ou non autorisé sont levées directement par la méthode `checkUserPriviledges()`.

Certaines erreurs sont directement gérées par Jersey ; le format de sérialisation ou les méthodes HTTP non supportés par le service Web conduisent respectivement aux erreurs HTTP 405 et 415.

L'erreur sera remonté au client via l'exception `WebApplicationException` :

```

@Service
@Scope("request")
@Path("comments/{componentId}/{contentId}")
public class CommentResource extends RESTWebService {

    @GET
    @Path("/{commentId}")
    @Produces(MediaType.APPLICATION_JSON)
    public CommentEntity getComment(@PathParam("commentId") String onComment) {
        checkUserPriviledges();
        try {
            ...
        } catch (CommentRuntimeException ex) {
            throw new WebApplicationException(ex, Status.NOT_FOUND);
        } catch (Exception ex) {
            throw new WebApplicationException(ex, Status.SERVICE_UNAVAILABLE);
        }
    }
}
...

```

3.2.3 L'état des ressources exposées

Chaque service Web expose un type particulier de ressources de Silverpeas. C'est l'état de ces ressources qui est échangé entre le service Web et les clients et ceci dans un format convenue. Dans HTTP, le format d'échange est appelé *média* et les objets échangés et donc transportés dans le corps des requêtes et des réponses HTTP sont appelés *entité*. On désignera donc par la suite sous ces termes respectivement le format d'échange et l'état des ressources exposées.

Les *entités* dans Silverpeas sont représentés par des objets à part entière. Ces objets qui représentent l'état d'une ressource exposée doivent satisfaire l'interface Exposable :

```

package com.silverpeas.rest;

import java.io.Serializable;
import java.net.URI;

/**
 * This interface is for qualifying the entities or functions that can be exposable as REST
 * resources through a REST web service.
 * All objects that can be exposable to the web must satisfy the contract defined by this interface.
 */
public interface Exposable extends Serializable {

    /**
     * Gets the URI at which this resource is published and can be accessed.
     * @return the web resource URI.
     */
    URI getURI();
}

```

Cette interface impose que certaines propriétés soient disponible.

Le *média* privilégié et recommandé dans Silverpeas est le JSON (Javascript Object Notation) car il permet d'échanger l'état des ressources directement avec des couches IHM Web via le javascript. Pour que Jersey puisse encoder ou décoder automatiquement les *entités*, les objets représentants celles-ci doivent utiliser les annotations JAXB (ce qui permet aussi l'encodage et le décodage en XML) :

```

@XmlRootElement
public class CommentEntity implements Exposable {

    private static final long serialVersionUID = 8023645204584179638L;
    @XmlElement(defaultValue = "")
    private URI uri;
    @XmlElement(defaultValue = "")
    private String id;
    @XmlElement(required = true)
    private String componentId;
    @XmlElement(required = true)
    private String resourceId;
    @XmlElement(required = true)
    private String text;
    @XmlElement(required = true)
    private CommentAuthorEntity author;
    @XmlElement(required = true, defaultValue = "")
    private String creationDate;
    @XmlElement(required = true, defaultValue = "")
    private String modificationDate;
    @XmlElement
    private boolean indexed = false;
    ...
}

```

Dans le cas où le *média* n'est pas directement supporté par Jersey, il est recommandé alors d'utiliser l'API d'export et d'import (définis dans le package Java `com.silverpeas.export` du module *lib-core*) avec les interfaces `Exporter` et `Importer`.

3.2.4 Tester le service Web REST

Pour tester un service Web, l'API fournit la classe de base des tests, `RESTWebServiceTest`, qui s'appuie sur le framework de tests de Jersey pour initialiser le contexte Web du service Web et les ressources qu'il requiert pour fonctionner. Cette classe fournit aussi des accesseurs sur des objets Silverpeas qui singent le comportement d'objets réels utilisés dans Silverpeas, comme `OrganizationController` ou `AccessController`.

Ces mocks et la classe de base des tests sur les services Web sont disponibles dans la bibliothèque Java *ws-test-core*. Il est donc nécessaire de l'inclure parmi les dépendances de votre projet (il est inclut par défaut par le module *web-core*) :

```

<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>com.silverpeas.core</groupId>
      <artifactId>ws-test-core</artifactId>
      <version>${core.version}</version>
      <scope>test</scope>
    </dependency>
    ...
  </dependencies>

```

Puis, chacune de vos tests doit étendre `RESTWebServiceTest` et dans le constructeur par défaut appeler celui du parent avec comme arguments le chemin du package Java où est défini le service Web à tester et le nom du fichier de configuration du contexte Spring à partir duquel les objets nécessaires à la préparation du test seront initialisés (ce fichier doit être placé à la racine de `src/test/resources/`) :

```

public abstract class BaseCommentResourceTest extends RESTWebServiceTest {

    protected static final String COMPONENT_INSTANCE_ID = "kmelia2";
    protected static final String CONTENT_ID = "1";
    protected static final String RESOURCE_PATH = "comments/" + COMPONENT_INSTANCE_ID + "/" +
CONTENT_ID;

    @Autowired
    private CommentServiceMock commentService;

    /**
     * Gets the comment service used in tests.
     * @return the comment service used in tests.
     */
    public CommentService getCommentService() {
        return commentService;
    }

    public BaseCommentResourceTest() {
        super("com.silverpeas.comment.web", "spring-comment-webservice.xml");
    }

    @Before
    public void checkCommentServiceMocking() {
        assertNotNull(commentService);
    }
}

```

Le fichier de configuration du contexte Spring doit comporter au moins l'instruction de détection des objets de tests définis dans ws-test-core et qui sont présents à partir du package Java com.silverpeas.rest :

```

<?xml version="1.0" encoding="UTF-8"?>
<!--

Copyright (C) 2000 - 2009 Silverpeas

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU Affero General Public License as
published by the Free Software Foundation, either version 3 of the
License, or (at your option) any later version.

As a special exception to the terms and conditions of version 3.0 of
the GPL, you may redistribute this Program in connection with Free/Libre
Open Source Software ("FLOSS") applications as described in Silverpeas's
FLOSS exception. You should have received a copy of the text describing
the FLOSS exception, and it is also available here:
"http://repository.silverpeas.com/legal/licensing"

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU Affero General Public License for more details.

You should have received a copy of the GNU Affero General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

-->

<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:p="http://www.springframework.org/schema/p"
  xmlns:ctx="http://www.springframework.org/schema/context"
  xsi:schemaLocation="
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">

  <ctx:annotation-config/>
  <ctx:component-scan base-package="com.silverpeas.rest"/>
  <ctx:component-scan base-package="com.silverpeas.comment.web"/>

```

Si le service Web à tester requiert que l'utilisateur soit authentifié et autorisé à accéder les ressources, il est nécessaire alors de créer cet utilisateur et de l'authentifier avant le déroulement du cas de test avec respectivement les méthodes `aUser()` et `authenticate(...)` définies dans `RESTWebServiceTest` :

```

@Before
public void createAUserAndAComment() {
    user = aUser();
    sessionKey = authenticate(user);
    theComment = aUser(user).
        commentTheResource(CONTENT_ID).
        inComponent(COMPONENT_INSTANCE_ID).
        andSaveItWithAsText("ceci est un commentaire");
}

```

L'authentification de l'utilisateur retourne une clé de session qui peut par la suite être utilisée dans l'entête `HTTP_SESSIONKEY` de la requête vers le service Web à tester :

```

@Test
public void getAComment() {
    WebResource resource = resource();
    CommentEntity entity = resource.path(RESOURCE_PATH + "/" +
        theComment.getCommentPK().getId()).
        header(HTTP_SESSIONKEY, sessionKey).
        accept(MediaType.APPLICATION_JSON).
        get(CommentEntity.class);
    assertNotNull(entity);
    assertThat(entity, matches(theComment));
}

```

Le paramètre HTTP_SESSIONKEY est défini dans la classe de base des services Web REST RESTWebService.

Le framework de tests de Jersey fournit ensuite un ensemble de méthodes pour simuler l'envoi de requêtes REST vers le service Web à tester et de récupérer automatiquement la réponse de celui-ci. Ces méthodes sont directement fournies par la classe WebResource qui est créé et initialisé par RESTWebServiceTest et qui peut être accédé par la méthode resource().

Tout appel vers le service Web à tester doit d'abord être préparé avec :

- le chemin où le service Web est publié :

```
resource.path(RESOURCE_PATH + "/" + theComment.getCommentPK().getId())
```

- la clé de session pour les utilisateurs authentifiés :

```
header(HTTP_SESSIONKEY, sessionKey)
```

- le média sous lequel l'état de la ressource à retourner est attendu :

```
accept(MediaType.APPLICATION_JSON)
```

- le média sous lequel l'état de la ressource est envoyé (pour les requêtes POST et PUT) :

```
type(MediaType.APPLICATION_JSON)
```

Puis il peut être envoyé :

- avec une méthode HTTP GET en précisant le type de l'objet attendu :

```
get(CommentEntity.class);
```

- avec une méthode POST en précisant le type de l'objet attendu dans la réponse et en passant l'état de la ressource à créer :

```
post(ClientResponse.class, theComment);
```

- avec une méthode HTTP PUT en précisant le type de l'objet attendu dans la réponse et en passant l'état avec lequel la ressource doit être mise à jour :

```
put(CommentEntity.class, aComment);
```

- avec une méthode HTTP DELETE :


```
delete();
```

– ...

Lorsqu'un cas de test attend une erreur du service Web testé, si la méthode du service Web ne renvoie pas explicitement d'objet Response, une exception `UniformInterfaceException` est levée et peut donc être attrapée pour être vérifiée :

```
@Test
public void getACommentWithADeprecatedSession() {
    WebResource resource = resource();
    try {
        resource.path(RESOURCE_PATH + "/3").header(HTTP_SESSIONKEY,
        UUID.randomUUID().toString()).
        accept(MediaType.APPLICATION_JSON).get(String.class);
        fail("A user shouldn't access the comment through an expired session");
    } catch (UniformInterfaceException ex) {
        int recievedStatus = ex.getResponse().getStatus();
        int unauthorized = Status.UNAUTHORIZED.getStatusCode();
        assertEquals(recievedStatus, is(unauthorized));
    }
}
```

Sinon, l'erreur peut être vérifiée directement via l'objet `ClientResponse` qui correspond à la réponse renvoyée par le service Web testé :

```
@Test
public void postANonAuthorizedComment() {
    denieAuthorizationToUsers();

    WebResource resource = resource();
    ClientResponse response = resource.path(RESOURCE_PATH).
    header(HTTP_SESSIONKEY, sessionKey).
    accept(MediaType.APPLICATION_JSON).
    type(MediaType.APPLICATION_JSON).
    post(ClientResponse.class, theComment);
    int recievedStatus = response.getStatus();
    int forbidden = Status.FORBIDDEN.getStatusCode();
    assertEquals(recievedStatus, is(forbidden));
}
```

Pour tester les erreur d'autorisation d'accès à une ressource, la classe de base `RESTWebService` propose la méthode `denieAuthorizationToUsers()` pour forcer l'échec à toute vérification d'autorisation de l'utilisateur par le service Web testé :

```
@Test
public void postANonAuthorizedComment() {
    denieAuthorizationToUsers();

    WebResource resource = resource();
    ClientResponse response = resource.path(RESOURCE_PATH).
    header(HTTP_SESSIONKEY, sessionKey).
    accept(MediaType.APPLICATION_JSON).
    type(MediaType.APPLICATION_JSON).
    post(ClientResponse.class, theComment);
    int recievedStatus = response.getStatus();
    int forbidden = Status.FORBIDDEN.getStatusCode();
    assertEquals(recievedStatus, is(forbidden));
}
```