REST
Un Survol des principaux concepts

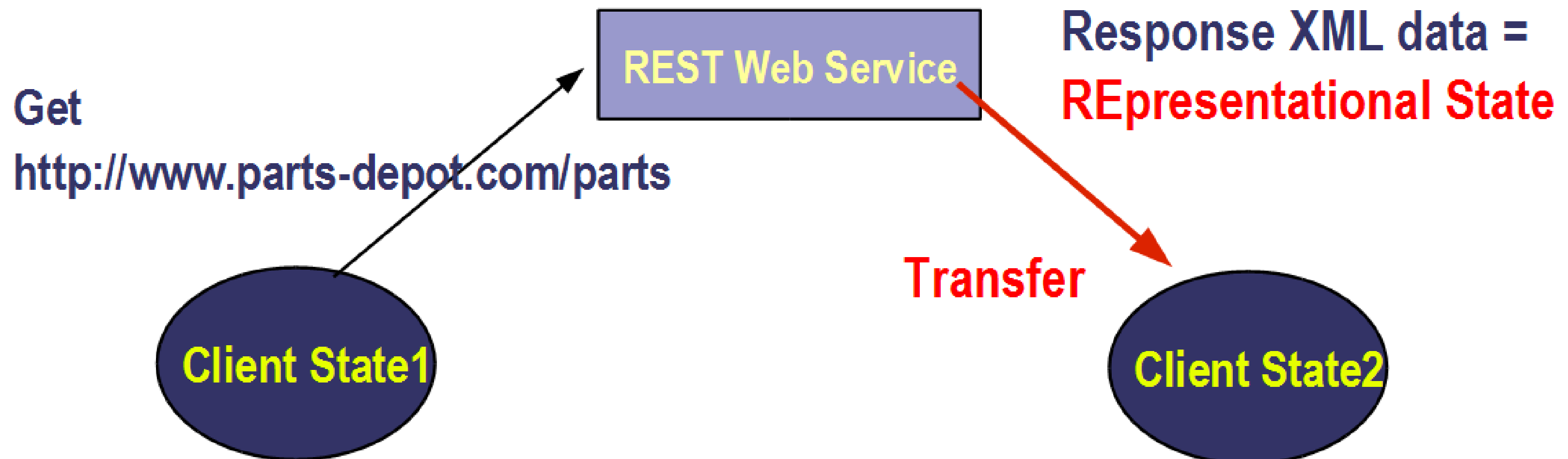
(basé essentiellement sur le support SUN par C.McDonald)

REST?

- **REST** (**REpresentational State Transfer**) est un style d’architecture pour les systèmes hypermédia distribués
- Créé par Roy Fielding en 2000 (thèse de doctorat)
- REST n’est pas un protocole (tel que HTTP) ou un format
- Style d'architecture particulièrement bien adapté au WWW mais n'est pas dépendant du Web.
  - Peut s'appliquer à d'autres protocoles d'application que HTTP.

REpresentational State Transfer (REST)

- L'URL c'est la Ressource
- GET pour afficher la page à partir du serveur
  - Transfert de l'état de la ressource sur le navigateur du client
- Les ressources sont accessibles à travers des liens hyperlink



Concepts clés

- Ressources (noms)
  - Identifiées par une URI, Exemple: http://www.parts-depot.com/parts
- Méthodes (verbes) afin de manipuler les ressources
  - Create, Read, Update, Delete
- Représentation est la manière de voir/échanger l'état de la ressource
  - Transfert de données et d'état entre le serveur et le client
  - XML, HTML, JSON...

Exemple

Request

```
GET /music/artists/beatles/recordings HTTP/1.1
Host: media.example.com
Accept: application/xml
```

Méthode

Ressource

Response

```
HTTP/1.1 200 OK
Date: Tue, 08 May 2007 16:41:58 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8

<?xml version="1.0"?>
<recordings xmlns="…">
  <recording>…</recording>
  …
</recordings>
```

Transfert d'état

Représentation

REST en 5 étapes

- Donner un ID pour chaque ressource
- Utiliser les méthodes standards d'HTTP
- Lier les ressources entre elles
- Choix entre multiples représentations
- Communication sans état (Stateless)

Etape 1: Donner un ID pour chaque ressource

- http://example.com/customers/1234
  - Le client num 1234 de la collection de clients
- http://example.com/products/4554
- http://example.com/customers/1234/orders/12
  - La commande num 12 du client num 1234

Etape 2: Utiliser les méthodes standards d'HTTP

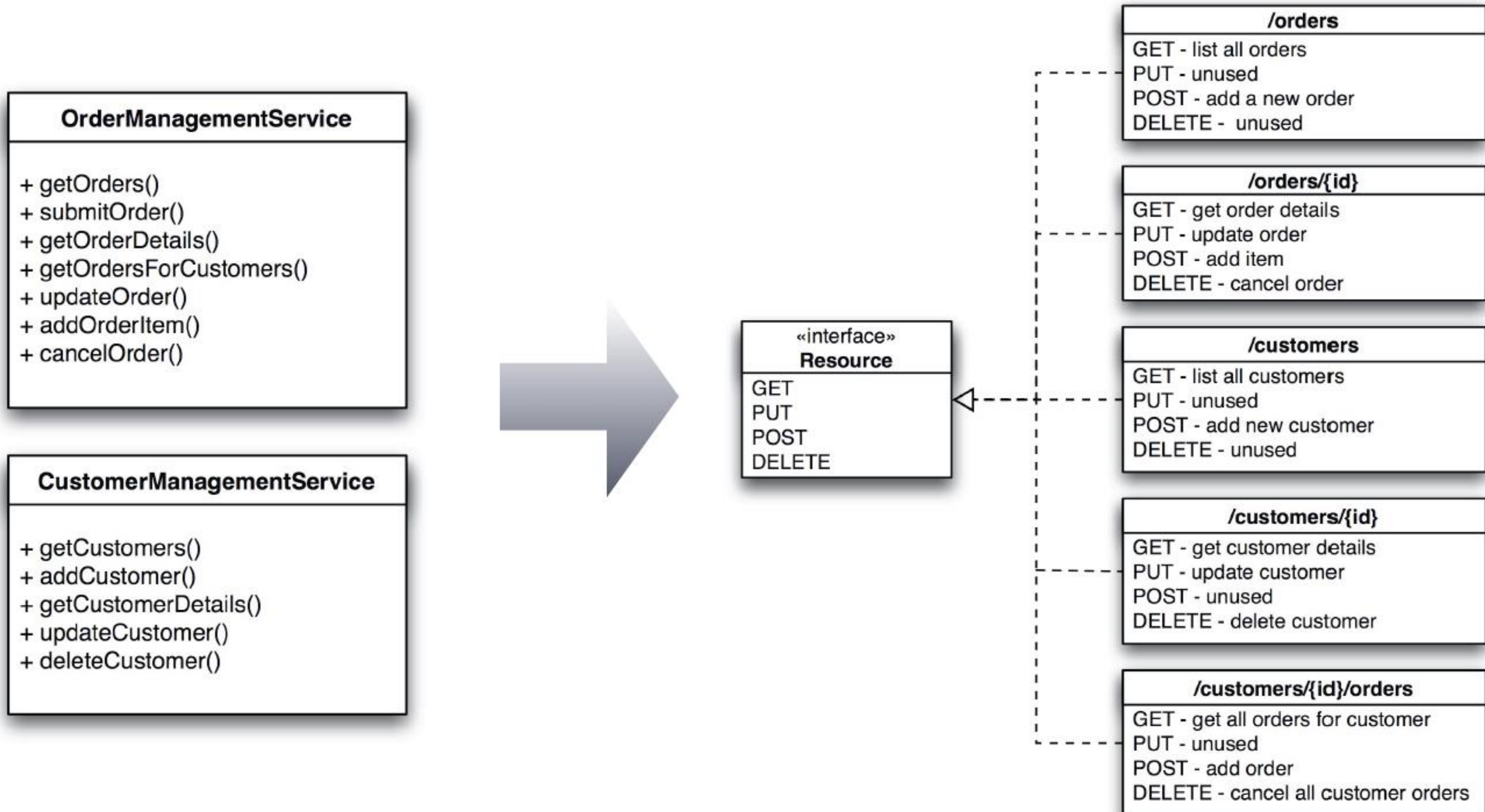
- GET: Lecture d'une information (ressource)
  - éventuellement déjà présente dans le cache
  - Sans effet de bord
    - Exp. **GET /toto/customers/1234**
- POST: Créer une nouvelle information (ressource) sans l'ID
  - Créer la ressource et la rajouter à une collection
  - Exp. POST **/toto/customers**
    - Ajoute le client spécifié dans le POSTDATA à la collection des clients.
    - L'opération retourne l'URI de la nouvelle ressource créée

Etape 2: Utiliser les méthodes standards d'HTTP

- PUT: Mise à jour /création d'une ressource avec un ID connu
  - Exp. PUT /toto/customers/1234
    - Remplace le client num 1234 avec une nouvelle version

- DELETE: Effacer une ressource
  - Exp. /toto/customers/1234
    - Efface le client num 1234 du système

Ce qui change dans la conception



Etape 3: Lier les ressources entre elles

- Permet au client de faire évoluer l’application d’un état à un autre en suivant des liens et en remplissant des formulaires

```
<order self=’http://example.com/orders/1234’>
	<amount>23</amount>
	<product ref=’http://example.com/products/4554’ />
	<customer ref=’http://example.com/customers/1234’ />
</order>
```

Etape 4: Choix entre multiples représentations

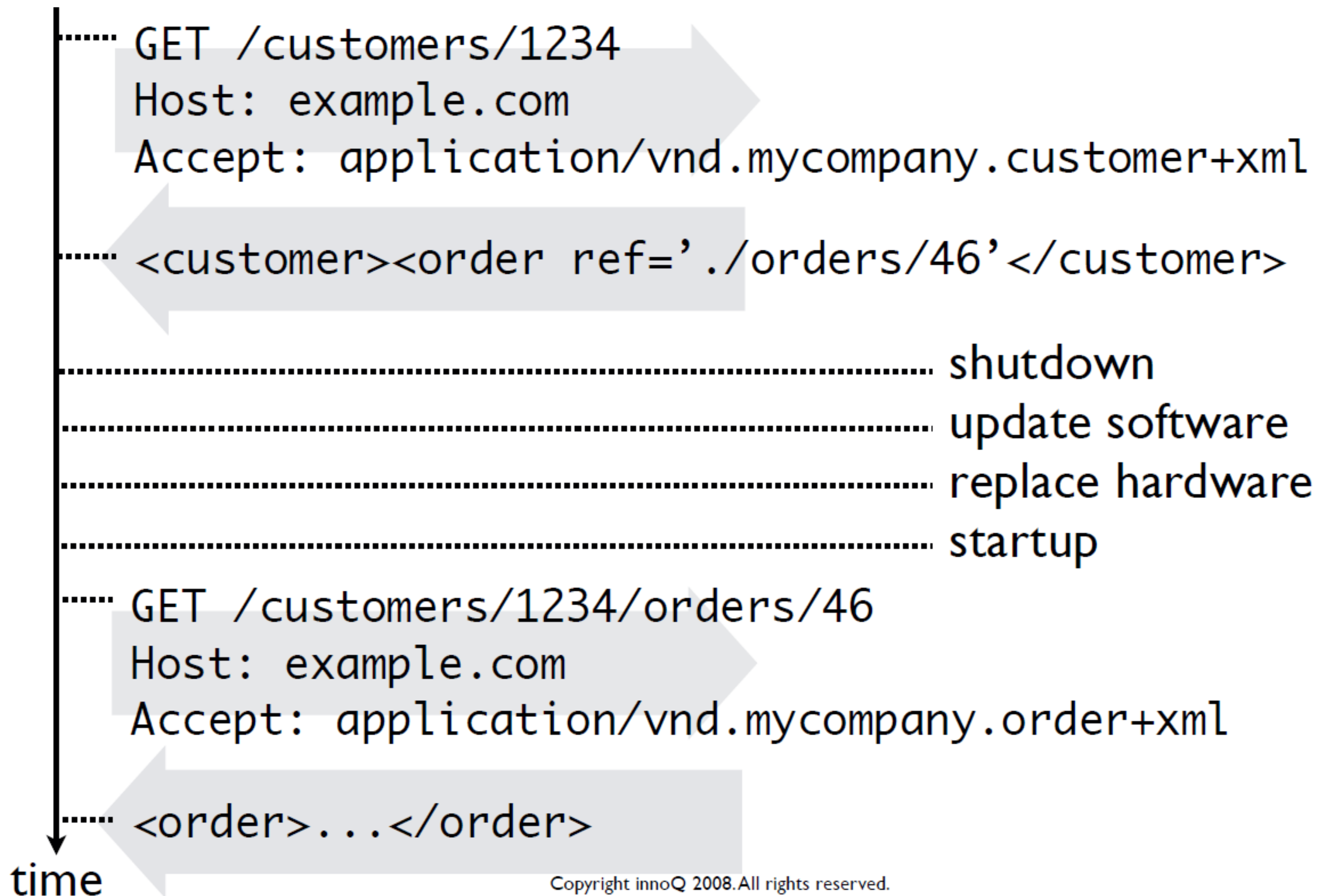
- Plusieurs formats possibles selon les besoins
  - XML, JSON, (x)HTML

```
// This method is called if TEXT_PLAIN is request
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello() {
return "Hello Jersey";
}

// This method is called if XML is request
@GET
@Produces(MediaType.TEXT_XML)
public String sayXMLHello() {
return "<?xml version=\"1.0\"?>" + "<hello> Hello Jersey" + "</hello>";
}

// This method is called if HTML is request
@GET
@Produces(MediaType.TEXT_HTML)
public String sayHtmlHello() {
return "<html> " + "<title>" + "Hello Jersey" + "</title>" + "<body><h1>" +
"Hello Jersey" + "</body></h1>" + "</html> ";
}
```

Etape 5: Communication sans état (Stateless)

- HTTP est Stateless (sans état)
- Tout ce qui est nécessaire pour traiter une demande est dans l'objet Request
- Le client est responsable de l'état de l'application
- Le serveur est responsable de l'état de la ressource
- Exp. Agence de voyage en ligne
  - Créer un voyage, définir l'itinéraire, le soumettre, etc.
  - Le tout est géré coté client et non pas sur la session du serveur

Etape 5: Communication sans état (Stateless)



Copyright innoQ 2008. All rights reserved.

REST: Principaux avantages

- Coté Serveur
  - Plus de passage à l’échelle (scalable)
  - Reprise après panne
  - Utilisation optimisée du cache
  - Couplage réduit
  - Fonctionne avec les infrastructures actuelles
  - Interface uniforme

- Coté Client
  - Liens bookmarkables (favoris)
  - Besoin d’un simple navigateur
  - Plusieurs langages supportés
  - Plusieurs choix de formats de données

REST

Un exemple

- Utilisation d’annotations JAX-RS

Etape 1: Donner un ID pour chaque ressource

- Une ressource => Classe POJO (Plain Old Java Object)
  - Pas d'interface requise!

- L'ID est défini par l'annotation **@Path**
  - Relative au contexte de déploiement
  - Peut être utilisée pour annoter la classe ou directement la méthode censée retourner la ressource

```
@Path("orders/{id}")          <- http://host/ctx/orders/12
public class OrderResource {

  @Path("customer")           <- http://host/ctx/orders/12/customer
  CustomerResource getCustomer(...) {...}
}
```

Etape 1: Donner un ID pour chaque ressource

- Comment mapper les URIs aux Classes:

```
@Path("/items")          Collection contenant les items du catalogue
public class Items {
    @Get
    public ItemsConverter get() {
        ... return new ItemsConverter(itemList);
    }
    @Path("{id}/")                        retourne 'item' selon l'id
    public ItemResource getItem(@PathParam("id")int id) {
        ... return itemResource;
    }
}
```

Etape 1: Donner un ID pour chaque ressource

- Deux manières de créer des sous-ressources

```
public class ItemResource {
    @Path("/items/{id}/")
    @GET
    public ItemConveter get(@PathParam("id") Long id) {
 .....
    }
```

```
@Path("/items/")
 public class ItemsResource {
 @Path("{id}/")
 public ItemResource getItemResource(@PathParam("id") Long id) { ...
  return resource;
 }
```

Etape 2: Utiliser les méthodes standards d'HTTP

- Annoter les classes de ressources avec les méthodes standards selon le besoin
  - @GET, @PUT, @POST, @DELETE

```
// For the browser
@GET
@Produces(MediaType.TEXT_XML)
public Todo getTodoHTML() {
  Todo todo = TodoDao.instance.getModel().get(id);
  if(todo==null)
    throw new RuntimeException("Get: Todo with " + id +  " not found");
  return todo;
}

@PUT
@Consumes(MediaType.APPLICATION_XML)
public Response putTodo(JAXBElement<Todo> todo) {
  Todo c = todo.getValue();
  return putAndGetResponse(c);
}

@DELETE
public void deleteTodo() {
  Todo c = TodoDao.instance.getModel().remove(id);
  if(c==null)
    throw new RuntimeException("Delete: Todo with " + id +  " not found");
}

private Response putAndGetResponse(Todo todo) {
  Response res;
  if(TodoDao.instance.getModel().containsKey(todo.getId())) {
    res = Response.noContent().build();
  } else {
    res = Response.created(uriInfo.getAbsolutePath()).build();
  }
  TodoDao.instance.getModel().put(todo.getId(), todo);
  return res;
}
```

Le @Path n'est pas donné dans l'exemple, il est défini avant la signature de la classe

Etape 2: Utiliser les méthodes standards d'HTTP

- Possibilité d'extraire les informations à partir des paramètres de la requête avec **@QueryParam**

```
@Path("/items/")
@ConsumeMime("application/xml")
public class ItemsResource {

   @GET
   ItemsConverter get(@QueryParam("start")
     int start) {
    ...
   }

   @Path("{id}/")
   ItemResource getItemResource(@PathParam("id") Long id){
   ...
   }

}
```

http://host/catalog/items/?start=0

http://host/catalog/items/123

Etape 3: Lier les ressources entre elles

- **UriInfo** donne l'information sur le contexte de déploiement, l'URI, et le chemin jusqu'à la ressource

- **UriBuilder** offre des facilités pour créer les URIs des ressources

```
@Context UriInfo i;
OrderResource r = ...
UriBuilder b = i.getBaseUriBuilder();
URI u = b.path(OrderResource.class).build(r.id);

List<URI> ancestors = i.getAncestorResourceURIs();
URI parent = ancestors.get(ancestors.size()-1);
```

Etape 4: Choix entre multiples représentations

- Annoter les méthodes ou bien les classes avec
  - **@ProduceMime, @ConsumeMime**

```
@GET
@ProduceMime({"application/xml","application/json"})
Order getOrder(@PathParam("order_id") String id) {
   ...
}
@GET
@ProduceMime("text/plain")
String getOrder(@PathParam("order_id") String id) {
   ...
}
```

Etape 4: Choix entre multiples représentations

Request

```
GET /music/artists/beatles/recordings HTTP/1.1
Host: media.example.com
Accept: application/xml
```

Accept HTTP header

Format

Response

```
HTTP/1.1 200 OK
Date: Tue, 08 May 2007 16:41:58 GMT
Server: Apache/1.3.6
Content-Type: application/xml; charset=UTF-8

<?xml version="1.0"?>
<recordings xmlns="…">
  <recording>…</recording>
  …
</recordings>
```

Content-type HTTP header

Representation

Etape 4: Choix entre multiples représentations

- JAX-RS peu automatiquement faire du Marshalling/ UnMarshaling entre les messages HTTP et les types Java. Support de:
  - **text/xml, application/xml, application/json - JAXB class**
  - ***/*** – byte[], InputStream, File, DataSource
  - **text/*** – String
  - **application/x-www-form-urlencoded** -MultivaluedMap<String, String>

- Il suffit d'annoter votre pojo avec @XmlRootElement

```java
package de.vogella.jersey.jaxb.model;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
// JAX-RS supports an automatic mapping from JAXB annotated class to XML and JSON
// Isn't that cool?
public class Todo {
  private String summary;
  private String description;
  public String getSummary() {
    return summary;
  }
  public void setSummary(String summary) {
    this.summary = summary;
  }
  public String getDescription() {
    return description;
  }
  public void setDescription(String description) {
    this.description = description;
  }


}
```

Etape 4: Choix entre multiples représentations

- Exemple complet

La classe de service annontée par JAX-RS

```
package de.vogella.jersey.jaxb;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import de.vogella.jersey.jaxb.model.Todo;

@Path("/todo")
public class TodoResource {
// This method is called if XMLis request
@GET @Produces({ MediaType.APPLICATION_XML,
MediaType.APPLICATION_JSON })
public Todo getXML() {
Todo todo = new Todo();
todo.setSummary("This is my first todo");
todo.setDescription("This is my first todo");
return todo; }
// This can be used to test the integration with the browser
@GET @Produces({ MediaType.TEXT_XML })
public Todo getHTML() {
Todo todo = new Todo();
todo.setSummary("This is my first todo");
todo.setDescription("This is my first todo");
return todo; }
}
```

La classe utilisée par JAXB pour créer la représentation JSON ou XML

```
package de.vogella.jersey.jaxb.model;

import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
// JAX-RS supports an automatic mapping from JAXB annotated class to XML and JSON
// Isn't that cool?
public class Todo {
  private String summary;
  private String description;
  public String getSummary() {
    return summary;
  }
  public void setSummary(String summary) {
    this.summary = summary;
  }
  public String getDescription() {
    return description;
  }
  public void setDescription(String description) {
    this.description = description;
  }


}
```

Etape 5: Communication sans état (Stateless)

- Une nouvelle instance est créée pour chaque requête
  - Réduit les problèmes de concurences
- Les sessions HTTP ne sont pas supportées
- Le développeur doit gérer l'état de l'application à travers les représentations

Et le WSDL dans tout ça?

- Pas nécessaire mais un format existe, WADL!!
  - WADL (Web Application Description Language)
  - https://wadl.dev.java.net/

```
<resources base="http://api.search.yahoo.com/NewsSearchService/V1/">
    <resource path="newsSearch">
        <method name="GET" id="search">
            <request>
                <param name="appid" type="xsd:string" style="query" required="true"/>
                <param name="query" type="xsd:string" style="query" required="true"/>
                <param name="type" style="query" default="all">
                    <option value="all"/>
                    <option value="any"/>
                    <option value="phrase"/>
                </param>
                <param name="results" style="query" type="xsd:int" default="10"/>
                <param name="start" style="query" type="xsd:int" default="1"/>
                <param name="sort" style="query" default="rank">
                    <option value="rank"/>
                    <option value="date"/>
                </param>
                <param name="language" style="query" type="xsd:string"/>
            </request>
            <response>
                <representation mediaType="application/xml" element="yn:ResultSet"/>
                <fault status="400" mediaType="application/xml" element="ya:Error"/>
            </response>
        </method>
    </resource>
</resources>
```

Conclusion

- Style architectural de plus en plus utilisé
- Défini comme étant le vrai WEB contrairement aux WS SOAP
- Jeunesse encore au niveau des standards de sécurité, transactions, etc.
- Eviter de généraliser!! Le tout REST ou le tout SOAP!!

Zoom sur JAX-RS 2.0

JAX-RS 2.0

- **JAX-RS 2.0** is a framework that helps you in writing the RESTful web services on the
- client side as well as on the server side. **Jersey 2.0** is the reference implementation
- of the JAX-RS 2.0 (JSR 339 specification). Along with the enhancements in Java EE 7,
- JAX-RS 2.0 has also been revised dramatically

Les principaux Jars

- asm-all-repackaged-2.2.0-b14.jar
- cglib-2.2.0-b14.jar
- guava-14.0.1.jar
- hk2-api-2.2.0-b14.jar
- hk2-locator-2.2.0-b14.jar
- hk2-utils-2.2.0-b14.jar
- javax.annotation-api-1.2.jar
- javax.inject-2.2.0-b14.jar
- javax.ws.rs-api-2.0.jar
- jersey-client-2.2.jar
- jersey-common-2.2.jar
- jersey-container-servlet-core-2.2.jar
- jersey-server-2.2.jar
- osgi-resource-locator-1.0.1.jar
- validation-api-1.1.0.Final.jar

Le web.xml

```
...........
<servlet>
<servlet-name>simpleJerseyExample</servlet-name>
<servletclass>
org.glassfish.jersey.servlet.ServletContainer</servletclass>
<init-param>
<param-name>jersey.config.server.provider.packages</paramname>
<param-value>org.lip6.myresourcepackage</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>simpleJerseyExample</servlet-name>
<url-pattern>/services/*</url-pattern>
</servlet-mapping>
...........
```

Le HelloWorld

```
package org.lip6.myresourcepackage;
import javax.ws.rs.Path;
import javax.ws.rs.Get;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.Produces;
/**
* helloWorld Root Resource
*/
@Path("helloWorld")
public class HelloWorldResource{
@GET
@PRODUCES(MediaType.TEXT_PLAIN)
public String greet(){
return "Hello World!!!";
}
}
```

Pour tester? => sur votre navigateur, ou Chrome postman, ou SoapUI ou curl, etc.

http://localhost:8080/projectName/services/helloWorld

Les annotations

- @Path("resource_path"): Inique le path pour accéder à la ressource. Il est relatif à l'URL :
@serveur/nomApplication/urlPattern dans le web.xml/ [@Path("resource_path")]*

- @PathParam: utilisé pour injecter des valeurs à partir de l'URL vers les paramètres de la méthode

- @GET

- @PUT

- @POST

- @DELETE

- @Produces(MediaType.TEXT_PLAIN): définit le type MIME renvoyé par la méthode annotée par les méthodes HTTP (CRUD)

- @Consumes(type):définit quel type MIME est consommé par la méthode

Les annotations

- @PathParam
- @QueryParam
- @MatrixParam
- @FormParam
- @BeanParam
- @DefaultValue
- @HeaderParam
- @CookieParam
- @Context

@Path

```
@Path("helloWorld")
public class HelloWorldResource {
        @GET
        @Produces("text/plain")
        public String sayHello() {
                return "Hello World!";
        }
}
```

Des variables peuvent se rajouter à l'URI

- au niveau de la classe :

```
@Path("/helloWorld/{name}")
public class HelloWorldResource {
@GET
@Produces("text/plain")
public String sayHello(@PathParam("name") String name) {
return "Hello, " + name;
}
}
```

- au niveau de la méthode:

```
@Path("/helloWorld")
public class HelloWorldResource {
@GET
@Produces("text/plain")
@Path("{name}")
public String sayHello(@PathParam("name") String name) {
return "Hello, " + name;
}
}
```

GET http://localhost:8080/JAXRSDemo/services/helloWorld/John

@Path avec Expression Régulière

```
@Path("/helloWorld")
public class HelloWorldResource {
@GET
@Produces("text/plain")
@Path("{name: ([a-zA-Z])*}")
public String sayHello(@PathParam("name") String name) {
return "Hello, " + name;
}
}
```

- Ok :
http://localhost:8080/JAXRSDemo/services/helloWorld/John
Output:
Hello, John
- Erreur:
http://localhost:8080/JAXRSDemo/services/helloWorld/John1987
Output:
HTTP Status 404, Not Found

Les opérations HTTP

- @GET: Lire une représentation d'une ressource. Aucun effet de bord
- @PUT: Utilisée pour l'update. Peut être utilisée pour la création si l'id de la ressource est défini par le client au lieu du Serveur (voir exemple POST, diapo suivante)
- @DELETE: pour supprimer une ressource

```
@DELETE
@Path("{name}")
public void
delete(@PathParam("name")String name)
{
System.out.println("DELETE: " + name);
}
```

Les opérations HTTP

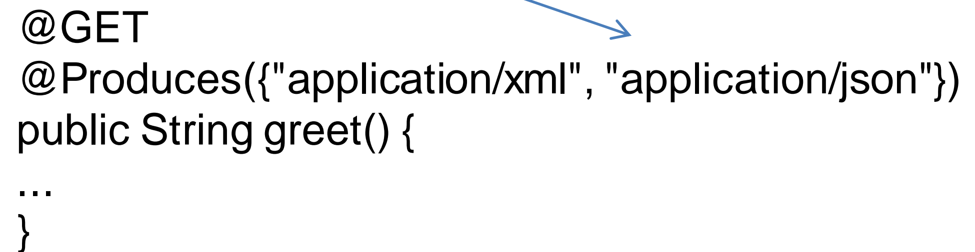
- @POST: pour la création d’une nouvelle ressource

```
@POST
 @Produces(MediaType.TEXT_HTML)
 @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
 public void newTodo(@FormParam("id") String id,
     @FormParam("summary") String summary,
     @FormParam("description") String description,
     @Context HttpServletResponse servletResponse) throws IOException {
   Todo todo = new Todo(id,summary);
   if (description!=null){
     todo.setDescription(description);
   }
   TodoDao.instance.getModel().put(id, todo);

   servletResponse.sendRedirect("../MainPage.html");
 }
```

@Produces & @Consumes

- Peut être fixée au niveau de la classe et redéfini au niveau de la méthode
- Peut prendre plusieurs types à la fois

```
@Path("/helloWorld")
@Produces("text/plain")
public class HelloWorldResource {
@GET
public String greet() {
...
}
@GET
@Produces("text/html")
public String greetUser() {
...
}
}
```



```
@GET
@Produces({"application/xml", "application/json"})
public String greet() {
...
}
```

Les annotations pour les paramètres

@PathParam

```
@GET
@Path("{name}")
public String getUserByName(@PathParam("name") String name) {
return name;
}
```

@QueryParam

```
@Path("/userService")
public class UserResource {
...
@GET
@Path("/queryParam")
public String getUser(@QueryParam("name") String name) {
System.out.println("Name: " + name);
return name;
}
...
}
```

@DefaultValue

```
@GET
@Path("/queryParam")
public String getUser (
@QueryParam("name")String name,
@DefaultValue("15") @QueryParam("age") String age) {
System.out.println("Name: " + name);
System.out.println("Age: " + age);
return name;
}
```

URI Pattern: /services/userService/queryParam?name=John
URI Pattern: /services/userService/queryParam?name=John&age=20

@MatrixParam

Exemple URI Pattern: **/service/getUserById/1;name=John;age=10**

```
@GET
@Path("/getUserById/{userId}")
public Response getUserById(
@PathParam("userId") String userId,
@MatrixParam("name") String name,
@DefaultValue("15") @MatrixParam("age") String age) {
return Response
.status(200)
.entity("Id: " + userId + ", Name: " + name + ", Age: " + age)
.build();
}
```

@FormParam

```
@POST
 @Produces(MediaType.TEXT_HTML)
 @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
 public void newTodo(@FormParam("id") String id,
     @FormParam("summary") String summary,
     @FormParam("description") String description,
     @Context HttpServletResponse servletResponse) throws IOException {
   Todo todo = new Todo(id,summary);
   if (description!=null){
     todo.setDescription(description);
   }
   TodoDao.instance.getModel().put(id, todo);

   servletResponse.sendRedirect("../MainPage.html");
 }
```

@BeanParam

- Permet l’injection de plusieurs paramètres (styles) dans le même Bean
- Objectif: mettre tous les paramètres d’une request dans un seul objet, le Bean

@BeanParam

```
public class UserBean {
@PathParam("id")
private String id;
@MatrixParam("name")
private String name;
@MatrixParam("age")
private String age;
@DefaultValue("No address provided")
@QueryParam("address")
private String address;
@HeaderParam("user-agent")
private String userAgent;
….
}
```

```
@Path("/beanResource")
public class BeanResource {
@GET
@Path("/getUserDetails/{id}")
public String getUser(@BeanParam UserBean userBean) {
return "User Bean: " + userBean.toString();
}
}
```

GET http://exampe.com/services/beanResource/getUserDetails/1;name=John;age=25?address=USA

@CookieParam

- Les cookies sont un types spécifique de HTTP headers
- Composés de paires nom/valeur
- Échangés entre client et serveur
- Utilisés pour maintenir la session et stockés d'autres infos ou préférences clients

```
@GET
@Path("/getCookies")
public String getCookies(@CookieParam("sessionid") int sessionId) {
return "Session Id: " + sessionId;
}
```

@CookieParam

- Possibilité de récupérer une instance de Cookie

```
@GET
@Path("/getCookies")
public String getCookies(@CookieParam("user-agent") Cookie userAgentCookie) {
return
"Name: " + userAgentCookie.getName() +
"Value: " + userAgentCookie.getValue() +
"Domain: " + userAgentCookie.getDomain() +
"Path: " + userAgentCookie.getPath() +
"Version: " + userAgentCookie.getVersion();
}
```

Method	Description
getName()	Corresponds to the string name of the cookie
getValue()	Corresponds to the string value of the cookie
getDomain()	Specifies the DNS name
getPath()	Corresponds to the URI path from where the request is being called
getVersion()	Defines the format of the cookie header

Notion de Subresource

- Ici aucune Method HTTP est déclarée sur la méthode getAddress(). La request est forwarder à la subresource i.e. la nouvelle instance AddressResource

- La bonne méthode à appeler sera automatiquement résolut par le framework selon le types attendu, le type consommé, le type de méthode HTTP.

```
@Path("/userService")
public class UserResource {
………..
@Path("/getAddress")
public AddressResource getAddress() {
return new AddressResource();
}
}
```

```
public class AddressResource {
@GET
public Response getUserName() {
return Response.status(200).entity("Address").build();
}
}
```

URI Pattern: http://localhost:8080/Chapter2/services/userService/getAddress

Notion de scope

- Par défaut les instances de ressources sont Request-scoped.

- Possibilité de définir un singleton

```
@Singleton
@Path("/printers")
public class PrintersResource {
....
}
```

Déploiement

- Option 1: utilisation de la classe abstraite Application

```
public class MainApplication extends Application {
@Override
public Set<Class<?>> getClasses() {
Set<Class<?>> s = new HashSet<Class<?>>();
s.add(HelloWorld.class);
…..
return s;
}
}
```

```
….
<servlet>
<servlet-name>Jersey Web Application</servlet-name>
<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
<init-param>
<param-name>javax.ws.rs.Application</param-name>
<param-value>com.example.MainApplication</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Jersey Web Application</servlet-name>
<url-pattern>/services/*</url-pattern>
</servlet-mapping>
…….
```

Déploiement

- Option 2: utilisation de la classe ResourceConfig

```
public class MainApplication extends
ResourceConfig{
public MainApplication() {
packages("com.example");
}
}
```

Possibilité de définir plusieurs
packages("com.example;com.example2;com.example3");

```
….
<servlet>
<servlet-name>Jersey Web Application</servlet-name>
<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
<init-param>
<param-name>javax.ws.rs.Application</param-name>
<param-value>com.example.MainApplication</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Jersey Web Application</servlet-name>
<url-pattern>/services/*</url-pattern>
</servlet-mapping>
…….
```

Déploiement

- Option 3: Aucune pré-configuration préalable n'est requise=> juste le mentionner dans le Web.xml

```
…….
<servlet>
<servlet-name>Jersey Web Application</servlet-name>
<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
<init-param>
<param-name>jersey.config.server.provider.packages</param-name>
<param-value>com.example ;com.example2;com.example3</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
<servlet-name>Jersey Web Application</servlet-name>
<url-pattern>/services/*</url-pattern>
</servlet-mapping>
```

Bibliographie et webographie

- http://fr.slideshare.net/confoo/introduction-la-scurit-des-webservices
- http://www.journaldunet.com/developpeur/tutoriel/xml/070502-xml-ws-security.shtml
- http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss
- http://www.fidens.fr/articles/-bonnes-pratiques-securite-des-web-services-70.html