

SOA
Restful Web services
Jersey

Qu'est ce qu'un service web ?

Service web = instance d'objet, ressource déployée sur Internet

Permettent à des applications (web, mobile, etc.) de faire appel à des fonctionnalités (via des objets) sur un réseau local ou Internet

Evolution des systèmes distribués, architecture SOA (architecture orientée service)

Technologie initiée par IBM et microsoft, puis normalisée par le W3C

Un service web = composant développé dans n'importe quel langage sur n'importe quelle plateforme.

Il doit pouvoir être invoqué par n'importe quel autre service

Pourquoi les web services ?

Auparavant, utilisation des RMI, Corba,...

=>utilisation de démons (services) particuliers

Avec les web services, utilisation de serveurs web quelquesoit la plateforme
les messages passent par le protocole HTTP

Le fait d'utiliser les serveurs web sur ports ouverts (80,8080,...) permet aux services web d'être accessibles facilement (pas de pb de parefeu)

Pourquoi les web services ?

Utilisation d'HTTP:

- Protocole Internet
- Beaucoup d'entreprises possèdent un serveur web
- Protocole généralement autorisé au niveau de parefeu
- Protocole disponible sur toutes les plateformes
- Mode non connecté

Requêtes/réponses des services doivent être sérialisées pour réutilisation

XML

JSON

autre, attention (pas recommandé)

Pourquoi les web services ?

Principalement 2 types d'architectures de services

- Services SOAP (cours suivant) SOAP = protocole basé sur XML
- Services REST (*Representational State Transfer => transfert sur HTTP (pas de protocole XML implicite)*)
 - Utilisation d'HTTP, sérialisation des données Appel direct via http
 - Stateless !!!,
 - Utilisation des méthode HTTP (a.k.a. verbes) PUT, GET, etc
 - De base, pas de protocole de sécurité, pas de norme particulière,
 - mais plus simple

RESTful Web Services

- * Appel de ressources (Web) via une URL contenant un nom de méthode et des paramètres
 - * URL peut avoir une grammaire complexe
- * Utilisation d'une méthode HTTP qui décrit de façon succincte la sémantique de l'appel

RESTful Web Services

- * Utiliser les bonnes méthodes HTTP:
 - * POST ajouter
 - * PUT modifier
 - * GET lire
 - * DELETE

RESTful Web Services

- * **HATEOAS, Hypermedia as the Engine of Application State,**
- * **Contrainte sur SW REST telle que:**
- * **Un client peut interagir complètement avec un SW => à tout moment il doit connaître les possibilités offertes par le services c.a.d. les ressources qu'il peut appeler**

RESTful Web Services

GET /account/12345 HTTP/1.1

Host: somebank.org

Accept: application/xml

...

HTTP/1.1 200 OK

Content-Type: application/xml

Content-Length: ...

<?xml version="1.0"?>

<account>

<account_number>12345</account_number>

<balance currency="usd">100.00</balance>

<link rel="deposit"

href="/account/12345/deposit" />

<link rel="withdraw"

href="/account/12345/withdraw" />

<link rel="transfer"

href="/account/12345/transfer" />

<link rel="close" href="/account/12345/close"

/>

</account>

Plus tard...

* HTTP/1.1 200 OK

* Content-Type: application/xml

* Content-Length: ...

*

* <?xml version="1.0"?>

* <account>

*

<account_number>12345</account_number>

* <balance currency="usd">-

25.00</balance>

* <link rel="deposit"

href="/account/12345/deposit" />

*

</account>

Introduction aux Services Web REST

- * Implémentation d'une API en Java = **JAX-RS** (Java **API** for **REST**ful Web **S**ervices) qui est intégrée à Java EE>6
- *
- * Décrite par les spécifications JSR 311 et JSR 339
- * Plusieurs frameworks basés sur JAX-RS
 - * : RESTEasy, CXF, Jersey (oracle)
 - * Jersey 2 (mai 2013) choisi dans ce cours (<http://jersey.java.net/>)
- * Possibilité d'utiliser Axis 2(Apache)
 - * <http://axis.apache.org/axis2/java/core/docs/rest-ws.html>



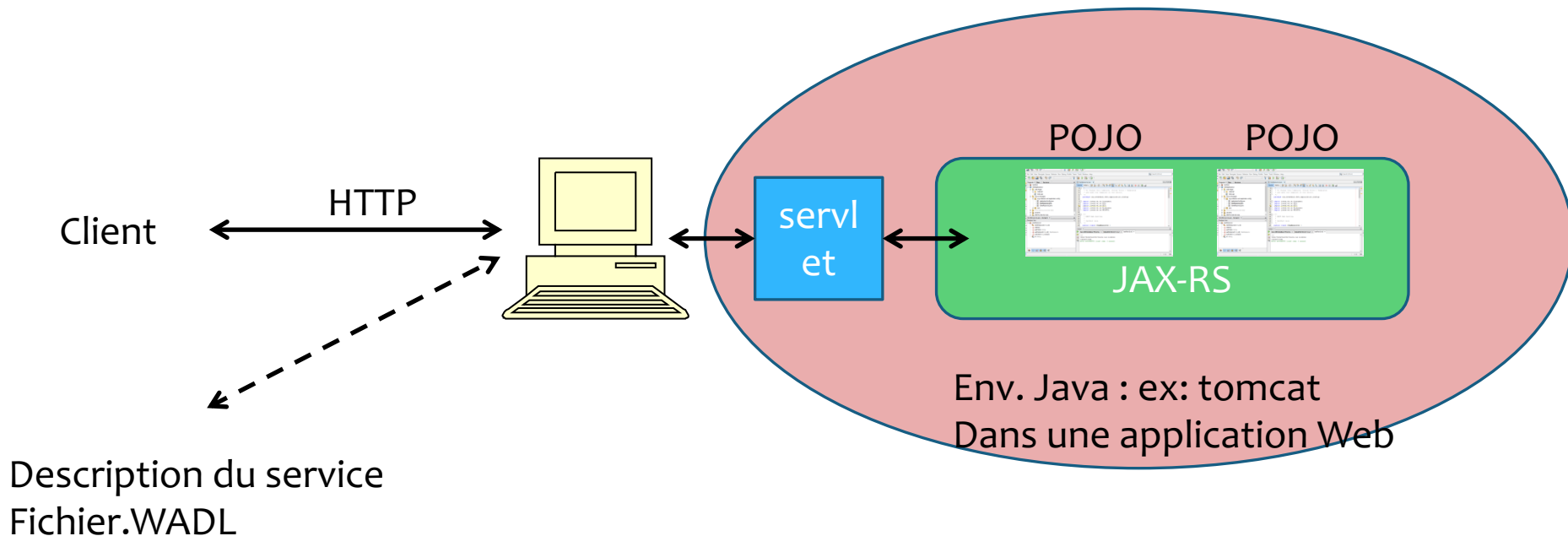
Introduction aux Services Web REST

- * JSR 339
- * Description
 - * des annotations à utiliser dans le code
 - * Du cycle de vie (via servlets)
 - * Des exceptions
 - * De la partie cliente
 - * Des URI
 - * Etc.



Architecture

- * Jersey => Développement basé sur l'implantation de POJO, annotés et sur des requêtes HTTP



REpresentational State Transfer (REST)

- * Création de service:
- * Principe:
 - * construire des ressources HTTP simple et stateless
 - * A tout moment, pour un service donné, on connaît les ressources disponibles
- * Création par approche Bottom/UP uniquement
 - * Création POJO+annotations
 - * Compilation et déploiement

Services Web REST avec Jersey

```
@Path("generic")
public class GenericResource {

    @GET
    @Produces("text/html")
    public String getResp() {

        return "<html><body>test<br>coucou\n</body></html>";
        //TODO return proper representation object
        // throw new UnsupportedOperationException();
    }
}
```

Chemin du service
vu comme une ressource

Type de requête HTTP

Type MIME de retour

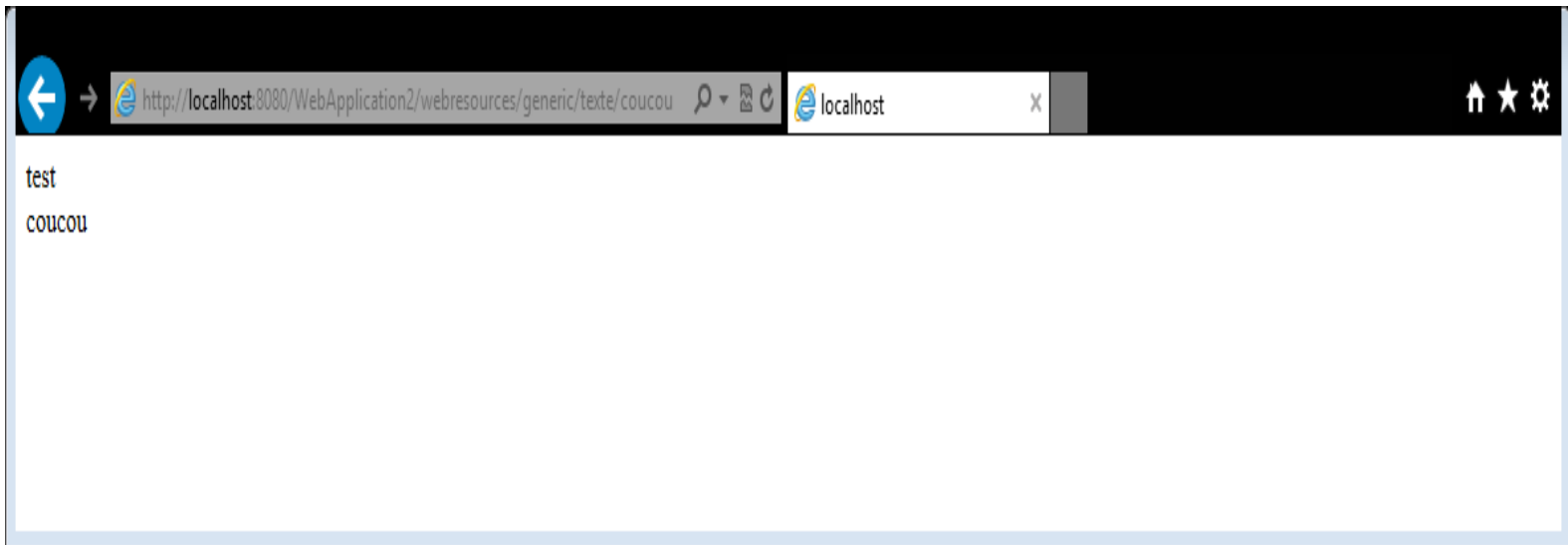
Exception éventuelle

Déploiement et appel

- * Déploiement en décrivant le service comme une Servlet (dans web.xml pour tomcat/Glassfish) (voir cours Servlet)
- * `<servlet>`
- * `<servlet-name>EssaisServlet</servlet-name>`
- * `<servlet-class>bdtest.EssaisServlet</servlet-class>`
- * `</servlet>`
- * `<servlet-mapping>`
- * `<servlet-name>EssaisServlet</servlet-name>`
- * `<url-pattern>/EssaisServlet</url-pattern>`
- * `</servlet-mapping>`
- * => les IDE tels que Netbeans ou Eclipse le font pour vous

Déploiement et appel

- * Appel avec méthode GET:



Les annotations

- * Choix de la méthode HTTP
 - * Définition des paramètres
 - * Définition des types de retour
 - * Sérialisation
-
- * Définis par des annotations dans le code

Les annotations

- * Annotation `@path`:

- * Classe java représentant un service REST doit être annotée par `@path` => définit une ressource Racine (root)

- * Path décrit une expression URI permettant d'appeler le service

- * Ex:

`http://localhost:8080/WebApplication2/webresources/g`
`eneric`

Contexte de l'application Web Contexte service



Les annotations

- * Sub ressource @Path
 - * Possibilité d'annoté les méthodes de la classe également
 - * url final = concaténation uri classe + uri méthode

```
@GET @Path("appel")  
@Produces("text/html")  
public String getResp() {
```

- * [http://localhost:8080/WebApplication2/webresources/generi
c/appe1](http://localhost:8080/WebApplication2/webresources/generi
c/appe1)

Les annotations

- * @Path peut également être complexe et donner des paramètres (appelées Template Parameters)
- * Distinction d'une expression par les balises {}

```
@GET @Path("appel/{texte}")  
@Produces("text/html")  
public String getResp(@PathParam("texte") String t) {
```

- * <http://localhost:8080/WebApplication2/webressources/generic/appel/coucou>

Les annotations

- * Code de méthode :

```
@GET
@Path("appel/{texte}")
@Produces("text/html")
public String getOb(@PathParam("texte") String texte) {

    return "<html><body>test<br>" + texte + "\n</body></html>";
    //TODO return proper representation object
    // throw new UnsupportedOperationException();
}
```

- * Appel:

<http://localhost:8080/WebApplication2/webresources/generic/appel/coucou>

La zone texte 

Exemple de méthode GET avec retour text/HTML

* Code de méthode :

```
@GET
@Path("appel/texte-{texte}-detail-{det}")
@Produces("text/html")
public String getOb(@PathParam("texte") String t,
@PathParam("det") String d) {

    return "<html><body>"+t+"<br>"+d+" \n</body></html>";
    //TODO return proper representation object
    // throw new UnsupportedOperationException();
}
```

* Appel:

<http://localhost:8080/WebApplication2/webresources/generic/appel/texte-coucou-detail-textecourt>

La zone texte



@HEAD, @Get, @Post, @Put, @Delete

- * Méthodes du services doivent être annotées par une méthode http pour traitement
- * Avec JAX-RS, il est possible d'utiliser @GET pour supprimer une ressource mais
 - * Ne respecte pas REST
 - * Manque de sens?
 - * Généralement description de méthodes CRUD (create, read, update, delete)
 - * @POST->creation ressource
 - * @GET->lecture
 - * @PUT->mise à jour
 - * @DELETE-> suppression de la ressource

Annotation supplémentaires pour les méthode

- * D'autres annotations pour gérer les paramètres des méthodes
- * Possibilité de faire des annotations sur des variables de types primitifs (sauf char), toute classe ayant un constructeur composé d'un paramètre String, classes ayant méthode statique valueOf(String)
- * Possibilité de placer une valeur par défaut avec @DefaultValue

Annotation supplémentaires pour les méthode

- * `@PathParam` -> extraire valeur dans URL sur le template parameters
- * `@QueryParam`-> extraire sur l'url par requête
- * `@FormParam`: extraction depuis formulaires HTML
- * `@CookieParam`: extraction depuis un paramètre de cookie
- * `@HeaderParam`: extraction de données de l'entête HTTP
- * `@context`: permet de récupérer contenu entête et cookies en même temps

Exemple avec @QueryParam

- * Exemple avec @PathParam (voir exemple précédent)

- * Exemple avec @QueryParam

@GET

@Path("appel/")

@Produces("text/html")

```
public String getOb(@DefaultValue("textedefault ") @QueryParam(" texte")  
String texte {
```

```
    return "<html><body>test<br>" + texte + "\n</body></html>";  
}
```

- * Appel:

[http://localhost:8080/WebApplication2/webresources/generic/appel/?texte=co
ucou](http://localhost:8080/WebApplication2/webresources/generic/appel/?texte=co
ucou)

si d'autres paramètres : suivi de ¶m2=value2 etc.

Codages des données envoyées et produites: @Consumes, @Produces

- * @Consumes utilisé pour spécifier le ou les types MIME qu'une méthode peut accepter
- * @Produces donne le ou les types MIME qu'une méthode peut produire
- * Annotations peuvent porter sur classes ou sur méthodes
- * Attention: si rien n'est défini, tous les types MIMES pourront être acceptés ou produits

@Produces

@GET

```
@Path("appel")
@Produces("text/html")
public String getHTMLob(@DefaultValue("textedefault ")
@QueryParam("texte") String texte {

    return "<html><body>test<br>" + texte + "\n</body></html>";
}
```

```
@Path("appel")
@Produces("text/xml")
public String getXMLob(@DefaultValue("textedefault ")
@QueryParam("texte") String texte {

    return "<?xml version='1.0'?>" + "<contenu>livre vide</contenu>";
}
```

@Produces

Mieux:

```
@Path("appel")
@GET
@Produces("application/xml")
public Response getXml(@DefaultValue("textedefault")
@QueryParam("texte") String texte ) {
```

```
Response response =
Response.status(200).type(MediaType.TEXT_XML).entity("<?
xml version=\\\"1.0\\\"
?>"+<contenu>"+texte+"</contenu>").build();
return response;
}
```

Opération sur les flux

- * JAX-RS facilite la sérialisation/désérialisation vers Type java
- * */* vers byte[]
- * Text/* vers String
- * Text/html, application /xml,..., vers JAXBElement
- * Et d'autres...

Manipulation de types personnalisés

- * Type String: voir exemples précédents
- * Egalement File etc.

- * Type personnalisé: possibilité d'utiliser des types personnalisés définis par un schéma XML
 - * Besoin d'un mapping entre XML et objet

 - * 1. définition d'une classe avec annotations de type JAXB @XmlRootElement, @XmlType
 - * 2. format du contenu de la requête au service en XML ou JSON
 - * 3. Les contenu des requêtes sont définis par les annotations @Produces et @Consumes (text/xml, application/xml, application/json et autres)

Manipulation de types personnalisés

Exemple Simple:

* 1. Définition de la classe POJO

`@XmlElement (name = "texte")`

```
Public Class Texte {
```

```
    Protected String contenu;
```

```
    Public String getContenu(){ return contenu;}
```

```
    Public void setContenu(String s){this.contenu=s;}
```

```
    Public String toString(){ return contenu;}
```

```
}
```


Utilisation d'un type personnalisé

```
@PUT
@Path("update")
@Consumes(" application/xml")
public Void uptexte(Texte t) {
    System.out.println(t.getContenu);
}
```

```
@GET
@Path("appel")
@Produces(" application/xml")
public Texte gettexte() {
    Texte t=new Texte();
    t.setContenu("bla bla");
    return t;
}
```

* Ou JSON : @Produces("application/json")

Utilisation d'un type personnalisé

Un autre exemple de ressource:

```
@XmlRootElement(name="customer")
@XmlAccessorType(XmlAccessType.FIELD)
//indique que tous les champs non statiques de la classe sont pris en
//compte
public class Customer {

    @XmlAttribute(required=true) protected int id;
    //attribut reste dans l'elt. XML en cours
    @XmlElement(required=true) protected String firstname;
    @XmlElement(required=true) protected String lastname;
    @XmlElement(required=true) protected Address address;
    @XmlElement(required=true) protected String email;
    @XmlElement (required=true) protected String phone; public
    Customer() { } // Getter and setter methods // ... }
```

Gestion du status HTTP

- * Code retour HTML retourné lorsqu'un service REST est appelé:
- * Réponse sans erreur
 - * Code retour 200 avec contenu
 - * Code retour 204 sans contenu
- * Réponse avec erreur
 - * De 400 à 599
 - * Ex: 404 not found
 - * Ex: 405, method not allowed

Gestion du status HTTP

- * Possibilité de construire des réponses plus précises et de modifier les codes
 - * Choisir un code retour
 - * Fournir des paramètres dans l'entête
- * Besoin d'utiliser le patron builder
 - * Plusieurs méthode retournant ResponseBuilder

Gestion du status HTTP

- * Exemples:
- * `ResponseBuilder ok()` => statut 200
- * `ResponseBuilder serverError()` => statut >400
- * `ResponseBuilder status(Response.Status)`: défini un statut précis
 - * Status: `Responses.NOT_FOUND`
 - * `Response.Status.OK`, ou chiffre

Gestion du status HTTP

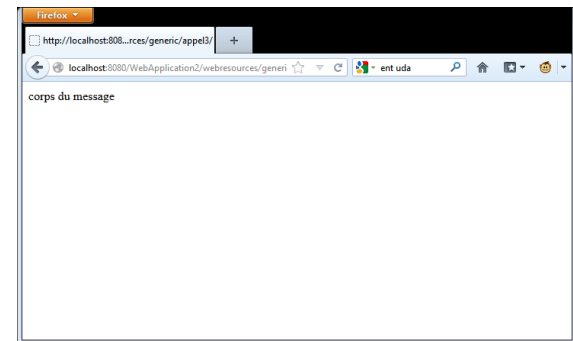
- * Méthodes de la classe Builder utiles:
 - * Response build() : crée une instance
 - * ResponseBuilder entity(Object o): modifie contenu du corps
 - * ResponseBuilder header(String, Object): modifie un paramètre de l'entête HTTP

Gestion du status HTTP

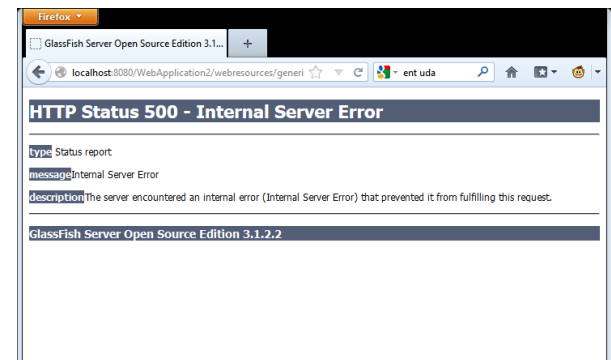
```
@Path("appel")
Public class Exemplerresponse {
@Path("response")
@GET
Public Response getText(){
Return Response
    .status(Response.Status.OK)
    .header("param1","test");
    .entity("corps du message");
    .build();
}
}
```

```
@GET
Public Response getText2(){
Return Response
    .serverError()
    .build();
}
}
```

Retour code 200



Retour code 500



Gestion des exceptions coté Serveur

- * Exception de base dans Jersey transformée en status HTTP:
WebApplicationException
 - * Plusieurs constructeurs (voir doc)

```
@GET
@Path("/images/{image}")
@Produces("image/*")
public Response getImage(@PathParam("image") String image) {
    File f = new File(image);

    if (!f.exists()) {
        throw new WebApplicationException(404);
    }

    String mt = new MimetypesFileTypeMap().getContentType(f);
    return Response.ok(f, mt).build();
}
```


Gestion des exceptions coté Serveur

- * Développement d'une Classe fille NotFoundException avec HTTP 404 (Not Found) status
- * Exemple d'appel :

```
throw new NotFoundException("Item, " + itemid + ", is not found");
```

Gestion des exceptions coté Serveur

```
1 public class NotFoundException extends WebApplicationException {
2
3     /**
4      * Create a HTTP 404 (Not Found) exception.
5      */
6     public NotFoundException() {
7         super(Responses.notFound().build());
8     }
9
10    /**
11     * Create a HTTP 404 (Not Found) exception.
12     * @param message the String that is the entity of the 404 response.
13     */
14    public NotFoundException(String message) {
15        super(Response.status(Responses.NOT_FOUND).
16            entity(message).type("text/plain").build());
17    }
18
19 }
```

Code client

Création de clients

- * Création de clients from scratch
- * Création de clients from scratch possible avec Axis2)
- * Génération de clients avec Netbeans 😊 mais limité

Création de clients Jersey

Classe Client : permet d'effectuer des appels

- * `Client client = ClientBuilder.newClient();`
- * `ClientConfig clientConfig = new ClientConfig();`
 - * Pour ajouter de la configuration (ex: SSL, etc.)

Création de clients Jersey

- * Classe WebTarget

- * Cibler la ressource HTTP
- * Identifier la ressource
- * Donner des parametres

1. WebTarget webTarget =
client.target(["http://example.com/rest"](http://example.com/rest));
2. WebTarget helloworldWebTarget =
resourceWebTarget.path("helloworld");
3. WebTarget helloworldWebTargetWithQueryParam =
helloworldWebTarget.queryParam("greeting", "Hi
World!");

Création de clients Jersey

- * Classe Invocation

- * Pour invoquer et récupérer une réponse

- * `Invocation.Builder invocationBuilder =`

- `helloworldWebTargetWithQueryParam.request(MediaType.TEXT_PLAIN_TYPE);`

- `invocationBuilder.header("some-header", "true");`

- * `Response response = invocationBuilder.get();`

- * `System.out.println(response.getStatus());`

- * `System.out.println(response.readEntity(String.class));`

Création de clients Jersey

- * Fluent support

```
Client client = ClientBuilder.newClient(new ClientConfig()  
    .register(MyClientResponseFilter.class)  
    .register(new AnotherClientFilter()));
```

```
String entity = client.target("http://example.com/rest")  
    .path("resource/helloworld")  
    .queryParams("greeting", "Hi World!")  
    .request(MediaType.TEXT_PLAIN_TYPE)  
    .header("some-header", "true")  
    .get(String.class);
```

Possibilité de
recupérer un
autre type que
String
Ex:
typeperso.class

Création de clients Jersey

- * Fluent support
- * Version courte:
- * Attention , on peut perdre pas mal de possibilités

```
String responseEntity = ClientBuilder.newClient()  
    .target("http://example.com").path("resource/rest")  
    .request().get(String.class);
```

Création de clients Jersey

- * Appel par get()
- * Appel par queryparam ou par l'url
 - * (normal)
 - * Reception d'un Objet deserialisé
- * Appel par put post etc.
 - * Possibilité d'envoyer un object qui est sérialisé
 - * Reception d'un Objet deserialisé

Création de clients Jersey

Exemple Code généré avec Netbeans :

```
public class Jclientexo4 {
    private WebTarget webTarget;
    private Client client;
    private static final String BASE_URI = "http://localhost:8080/restws-war/webresources";

    public Jclientexo4() {
        client = javax.ws.rs.client.ClientBuilder.newClient();
        webTarget = client.target(BASE_URI).path("exo4");
    }

    public <T> T getXml(Class<T> responseType) throws ClientErrorException {
        WebTarget resource = webTarget;
        resource = resource.path("appel")
            .queryParams("texte", "coucou");

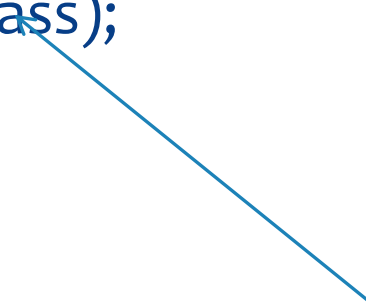
        return resource.request(javax.ws.rs.core.MediaType.APPLICATION_XML).get(responseType);
    }
}
```

Création de clients Jersey

Exemple Code généré avec Netbeans :

```
Jclientexo4 c= new Jclientexo4();  
String s=c.getXml(String.class);  
System.out.println(s);
```

Possibilité de
changer par classe
perso



Création de clients Jersey

2eme exemple :Code du service

```
public Exo5jsonResource() {  
    }  
  
    /**  
     * Retrieves representation of an instance of ws.Exo5jsonResource  
     * @return an instance of java.lang.String  
     */  
    @Path("appel")  
    @GET  
    @Produces("application/json")  
    public Texte getJson() {  
        Texte t= new Texte();  
        t.setContenu("json test");  
        return t;  
    }  
}
```

Création de clients Jersey

```
@Path("appelput")
```

```
@PUT
```

```
@Consumes("application/json")
```

```
@Produces("application/json")
```

```
public Texte putJson(Texte t) {
```

```
try[{
```

```
    t.setContenu(t.getContenu()+"passé par là");
```

```
    return t;
```

```
} catch (Exception e) {throw new WebApplicationException(404);} //gestion des erreurs  
coté Service !
```

```
}
```

```
}
```

Création de clients Jersey

Code du client (code généré avec Netbeans):

```
public Jclientexo5() {
    client = javax.ws.rs.client.ClientBuilder.newClient();
    webTarget = client.target(BASE_URI).path("exo5");
}

public <T> T getJson(Class<T> responseType) throws ClientErrorException {
    WebTarget resource = webTarget;
    resource = resource.path("appel");
    return
resource.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).get(responseType);
}

public Response putJson(Texte ent ) throws ClientErrorException {
    WebTarget resource = webTarget;
    resource = resource.path("appelput");
    return
resource.request(javax.ws.rs.core.MediaType.APPLICATION_JSON).put(Entity.entity(ent,
MediaType.APPLICATION_JSON));
}
```

Création de clients Jersey

```
//code du client (à utiliser dans le main())
```

```
try{
  Jclientexo5 c2= new Jclientexo5();
  Texte t=c2.getJson(Texte.class);
  System.out.println(t.toString());
}catch(ClientErrorException e){//to do}

Texte tinput=new Texte();
tinput.setContenu("second test");
Response r=c2.putJson(tinput);
if (r.getStatus()==200)
{
  Texte t3=(Texte) r.readEntity(Texte.class);
  System.out.println(t3.getContenu());
}
```

Gestion des
erreurs

Toujours à faire

Soit par status
Soit par exception

Sécurité, coté client

- * Authentification/ chiffrement

- * SSL

- * Définir un SSLContext ssl

- *

```
SSLContext ctx = SSLContext.getInstance("SSL"); ctx.init(null, myTrustManager, null); config.getProperties().put(HTTPSProperties.PROPERTY_HTTPS_PROPERTIES, new HTTPSProperties(hostnameVerifier, ctx));
```

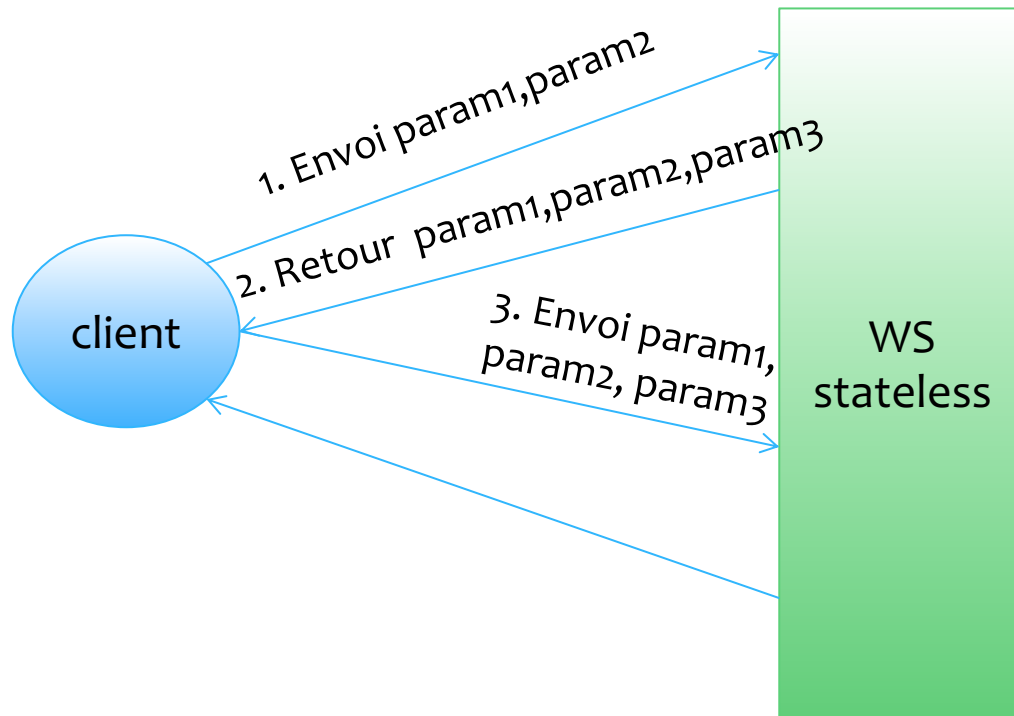
- * **Http Authentication (basicAuth et autre)**

- * HttpAuthenticationFeature

- *

```
HttpAuthenticationFeature feature = HttpAuthenticationFeature.basic("user", "superSecretPassword");
```

Rest = stateless



Pour construire une session, on peut passer tous les paramètres dans les requêtes successives, soit utiliser une base de donnée,...

Rest = stateless

- Avec Jersey, plusieurs scopes :

Scope request : mode normal

scope singleton

une seule instance de ressource entre tous les clients par jax-rs application

Annotation @Singleton

Spring boot

Quelques mots sur Spring boot

- * Micro framework conçu pour simplifier le démarrage et le développement de nouvelles applications Spring
- * Permet de créer des services Web Rest
- * Fonctionne également par annotations

Quelques mots sur Spring boot

- * Mise en place de l'application Spring

```
package hello;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class HelloWorldConfiguration { public static void main(String[] args)
{ SpringApplication.run(HelloWorldConfiguration.class, args);
}}}
```

SpringApplication.run() -> permet de lancer une appli Web

Quelques mots sur Spring boot

- * Implémentation classe métier

```
package hello;
public class Greeting {
    private final long id;
    private final String content;
    public Greeting(long id, String content) {
        this.id = id;
        this.content = content; }

    public long getId() { return id; }
    public String getContent() { return content; } }
```

Quelques mots sur Spring boot

- * En Spring, les services sont des contrôleurs

```
package hello;  
import ...
```

```
@Controller
```

```
@RequestMapping("/hello-world")
```

```
public class HelloWorldController {  
    private static final String template = "Hello, %s!";  
    private final AtomicLong counter = new AtomicLong();
```

```
@RequestMapping(method=RequestMethod.GET)
```

```
    public @ResponseBody Greeting sayHello(@RequestParam(value="name",  
        required=false, defaultValue="Stranger") String name) {  
        return new Greeting(counter.incrementAndGet(), String.format(template, name));  
    }  
}
```

Service récupère une requête GET pour /hello-world et retourne une instance de Greeting

Quelques mots sur Spring boot

- * En Spring, les services sont des contrôleurs, l'annotation `@SpringBootApplication` inclut une annotation `Scan` qui recherche automatiquement tous les contrôleurs
- * Code précédent renvoi du JSON par défaut
- * Contrôleur persiste ! (pas vraiment stateless...)
- * Autre exemple d'annotation Mapping (URLS)

```
@GetMapping("/students/{studentId}/courses/{courseId}") public Course  
retrieveDetailsForCourse(@PathVariable String studentId, @PathVariable String  
courseId) {
```

Et les micro-services ?

- * Microservices = services fonctionnels précis conçus pour réaliser parfaitement une seule chose.
- * *Chaque service est élastique, résilient, composable, minimal et complet*
- * Application = composition de micro-services
- * Concept qui s'adapte bien aux conteneurs (Docker, etc.)

Et les micro-services ?

- * Plusieurs frameworks légers:
- * comme [Dropwizard](#), [Spring Boot](#), [Spotify Apollo](#), [Spark](#) (Java), [Kumuluzee](#) (J2EE), [Flask](#) (Python), [Sinatra](#) (Ruby) ou [Vert.x](#) (Polyglotte).
- * Micronaut
- * Quakus

Micronaut



- * <https://micronaut.io/>
- * Framework de micro-services avec support natif de plateformes de cloud
 - * Utilise GraalVM
 - * Génère des fichiers de déploiement docker

Micronaut



* Exemple Service

```
@Controller("/cont")  
public class ContController {
```

```
    @Get(uri="/hello", produces="text/plain")  
    public String index() {  
        return "Example Response";  
    }  
}
```

```
    @Get(uri="/savegetEntity/{texte}", produces="text/html")  
    public String savegetEntity(@PathVariable String texte) {  
        ...]  
    }
```

Micronaut



- * Gestion des erreurs

- * Annotation `@error`

```
@Error public HttpResponse<JsonError> jsonError(HttpRequest  
request, JsonParseException jsonParseException) {...}
```

- * ExceptionHandler;

```
public class OutOfStockException extends RuntimeException { }
```

- * Ou créer des réponses HTTP

```
return HttpResponse.status(HttpStatus.UNAUTHORIZED).body(m);
```

Micronaut



Scopes

[@Singleton](#)

Singleton scope indicates only one instance of the bean should exist

[@Context](#)

Context scope indicates that the bean should be created at the same time as the ApplicationContext (eager initialization)

[@Refreshable](#)

@Refreshable scope is a custom scope that allows a bean's state to be refreshed via the /refresh endpoint.

[@RequestScope](#)

@RequestScope scope is a custom scope that indicates a new instance of the bean is created and associated with each HTTP request

En PHP?

- * Voir cours en janvier
- * Création de services simples avec
 - * Le Framework Slim (<http://www.slimframework.com/>)
 - * Le Framework Silex (<http://silex.sensiolabs.org/>)
- * Création de clients avec Guzzle , client type cURL (<http://guzzlephp.org/>)