

# SOA – Services web REST

# Développer des services web REST

# avec Java : JAX-RS

Mickaël BARON – 2011 (Rév. Février 2019)

<mailto:baron.mickael@gmail.com> ou <mailto:baron@ensma.fr>



@mickaelbaron



mickael-baron.fr

# Licence

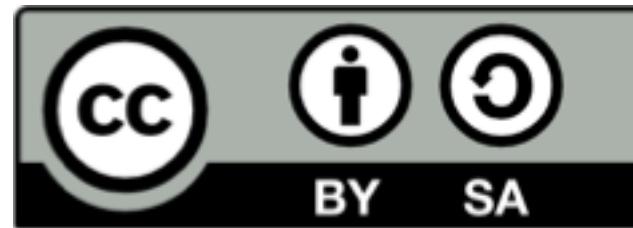
---

## Creative Commons

*Contrat Paternité*

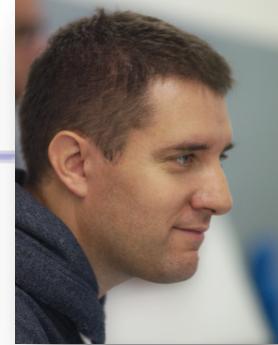
*Partage des Conditions Initiales à l'Identique*

2.0 France



<http://creativecommons.org/licenses/by-sa/2.0/fr>

## À propos de l'auteur ...



### ➤ Mickaël BARON

### ➤ Ingénieur de Recherche au LIAS

➤ <http://www.lias-lab.fr>



➤ Équipe : Ingénierie des Données et des Modèles



@mickaelbaron

➤ Responsable des plateformes logicielles, « coach » technique

### ➤ Responsable Rubriques Java de Developpez.com

➤ Communauté Francophone dédiée au développement informatique

➤ <http://java.developpez.com>



➤ 4 millions de visiteurs uniques et 12 millions de pages vues par mois

➤ 750 00 membres, 2 000 forums et jusqu'à 5 000 messages par jour

# Plan du cours

- Généralités JAX-RS
- Premier service web JAX-RS
- Développement serveur
  - Ressources : *@Path*
  - Méthodes : *@POST*, *@PUT*, *@DELETE* et *@GET*
  - Représentation : gestion du contenu et *Response*
- Développement client et test d'intégration
- Déploiement



# Déroulement du cours

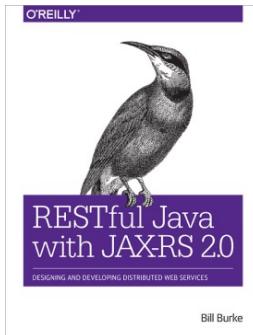
- Pédagogie du cours
  - Illustration avec de nombreux exemples qui sont disponibles à l'adresse *http://mbaron.developpez.com/soa/jaxrs*
  - Des bulles d'aide tout au long du cours
  - Survol des principaux concepts en évitant une présentation exhaustive
- Logiciels utilisés    
  - Navigateur web, Eclipse, Tomcat, Maven 3
  - Exemples « Mavenisés » indépendant de l'environnement de dév.
- Pré-requis
  - Schema XML, JAXB, Introduction services web
- Exemples
  - *https://github.com/mickaelbaron/jaxrs-examples*

# Ressources

---

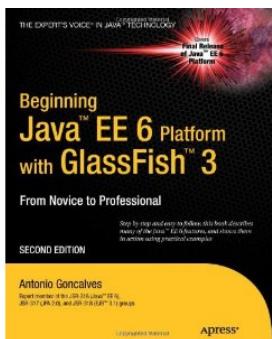
- [jersey.github.io](http://jersey.github.io)
- [jcp.org/en/jsr/detail?id=370](http://jcp.org/en/jsr/detail?id=370)
- [github.com/eclipse-ee4j/jaxrs-api](http://github.com/eclipse-ee4j/jaxrs-api)
- [github.com/eclipse-ee4j/jersey](http://github.com/eclipse-ee4j/jersey)
- [www.vogella.com/tutorials/REST/article.html](http://www.vogella.com/tutorials/REST/article.html)
- [www.baeldung.com/jax-rs-spec-and-implementations](http://www.baeldung.com/jax-rs-spec-and-implementations)
- [www.baeldung.com/jersey-test](http://www.baeldung.com/jersey-test)
- [www.baeldung.com/jersey-jax-rs-client](http://www.baeldung.com/jersey-jax-rs-client)
- [journal.utem.edu.my/index.php/jtec/article/view/3750](http://journal.utem.edu.my/index.php/jtec/article/view/3750)
- [docs.oracle.com/javaee/7/tutorial/partwebsvcs.htm](http://docs.oracle.com/javaee/7/tutorial/partwebsvcs.htm)
- [www.indestructiblevinyl.com/2016/07/23/logging-with-jersey-and-maven.html](http://www.indestructiblevinyl.com/2016/07/23/logging-with-jersey-and-maven.html)
- [yatel.kramolis.cz/2013/11/how-to-configure-jdk-logging-for-jersey.html](http://yatel.kramolis.cz/2013/11/how-to-configure-jdk-logging-for-jersey.html)

# Ressources : bibliothèque



## ➤ RESTful Java

- Auteur : Bill Burke
- Éditeur : Oreilly
- Edition : Nov. 2013 - 392 pages - ISBN : 9781449361341



## ➤ Beginning Java EE 6 Platform With GlassFish 3

- Auteur : Antonio Goncalves
- Éditeur : Apress
- Edition : Août 2010 - 536 pages - ISBN : 143022889X

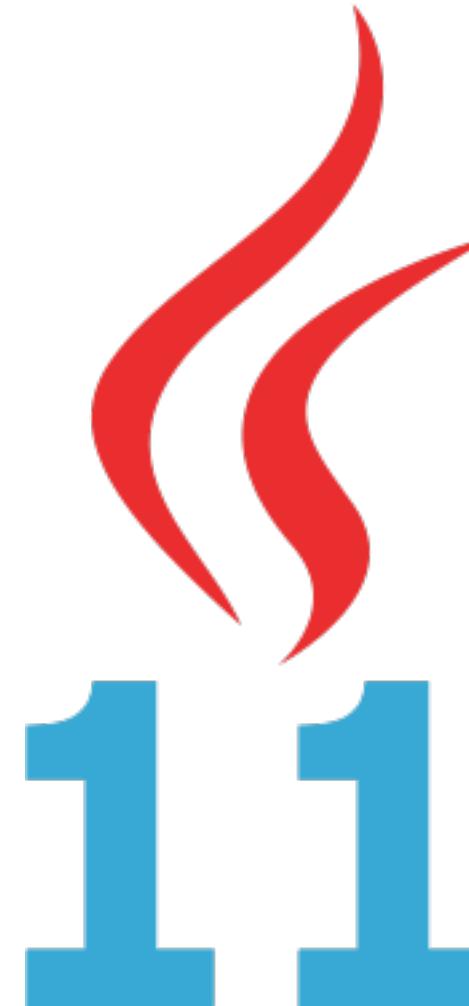


## ➤ RESTful Java Web Services

- Auteur : Jose Sandoval
- Éditeur : PACKT Publishing
- Edition : Nov. 2009 - 256 pages - ISBN : 1847196462

# Version Java supportée par ce support de cours

---



- Pas de gestion des modules dans les exemples (utilisation du classpath)

# Généralités : développement de services web REST

- Ce cours s'intéresse au développement des services web de type **REST**
  - Côté Serveur : code pour le traitement du service web
  - Côté Client : code qui permet d'appeler un service web
- La majorité des langages de programmation orientés web supportent le développement de services web REST
  - Java, PHP, C#, C++...
- Ce cours se limite au langage Java
- Différents *frameworks* de développement de services web
  - Ceux qui respectent la spécification JAX-RS (détailler après)
  - Autres ...



# Généralités JAX-RS : la spécification

- JAX-RS est l'acronyme Java API for RESTful Web Services
- Décrise par les JSR 339 (2.0) et JSR 370 (2.1)
- Version courante de la spécification est la 2.1
- Code source API : *github.com/eclipse-ee4j/jaxrs-api*
- Depuis la version 1.1, JAX-RS fait partie intégrante de la spécification Java EE 6 au niveau de la pile service web
- Cette spécification décrit la mise en œuvre de services web REST côté serveur et client
- Le développement des services web REST repose sur l'utilisation de classes Java et d'annotations

# Généralités JAX-RS : les implémentations

- Différentes implémentations de la spécification JAX-RS sont disponibles
- **JERSEY** : implémentation de référence fournie par Glassfish
  - Site projet : *jersey.github.io*
- **CXF** : fournie par Apache, la fusion entre **XFire** et **Celtix**
  - Site projet : *cxf.apache.org*
- **RESTEasy** : fournie par Widfly
  - Site projet : *resteasy.github.io*
- **RESTlet** : un des premiers framework implémentant REST pour Java
  - Site projet : *restlet.com*

# Généralités JAX-RS : les implémentations

- Comparaison sur les performances des implémentations

➤ [journal.utm.edu.my/index.php/jtec/article/view/3750](http://journal.utm.edu.my/index.php/jtec/article/view/3750)

- Dans la suite du support de cours nous utiliserons

l'implémentation de référence **JERSEY**



- Version actuelle **2.27** respectant la spécification JAX-RS 2.1
- Intégrée dans Glassfish depuis Java EE 6
- Site projet : [jersey.github.io](https://jersey.github.io)
- Code source : [github.com/eclipse-ee4j/jersey](https://github.com/eclipse-ee4j/jersey)

# Généralités JAX-RS : les implémentations et Maven

- Si vous utilisez un serveur d'application Java EE

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>2.1</version>
    <scope>provided</scope>
</dependency>
```

Nécessaire pour  
l'accès aux APIs  
(pour la  
compilation)

- Si vous utilisez un serveur d'application non Java EE

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

Serveur qui  
supporte  
Servlet 3.x

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-servlet-core</artifactId>
    <version>${jersey.version}</version>
</dependency>
```

Serveur qui  
supporte  
Servlet 2.x

# Généralités JAX-RS : fonctionnement

Développement de clients dans des langages différents



.NET



PHP



JAVA



Différentes APIs possibles pour la gestion du client en Java

**Couche Client**

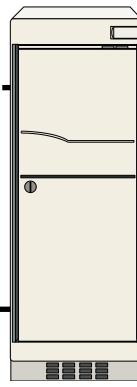
Description du service web permettant de générer la partie cliente

Open API

WADL

RAML

HTTP



Serveur Web

Servlet

**JAX-RS**

Conteneur Java



Classes JAVA annotées implémentant le service web

Utilisation du service web par envoi / réception de contenu HTTP

**Couche Serveur**



Approche Bottom / Up

## Généralités JAX-RS : développement

- Le développement de services web avec JAX-RS est basé sur des POJO (**P**lain **O**ld **J**ava **O**bject) en utilisant des annotations spécifiques à JAX-RS
- Pas description requise dans des fichiers de configuration
- Seule la configuration de la Servlet « JAX-RS » est requise pour réaliser le pont entre les requêtes HTTP et les classes Java annotées
- Un service web REST est déployé dans une application web

# Généralités JAX-RS : développement

- Contrairement aux services web étendus il n'y a pas de possibilité de développer un service REST à partir du fichier de description WADL
- Seule l'approche **Bottom / Up** est disponible
  - Créer et annoter un POJO
  - Compiler, Déployer et Tester
  - Possibilité d'accéder au document WADL
- Le fichier de description WADL est généré automatiquement par JAX-RS (exemple : *http://host/context/application.wadl*)
- Possibilité d'utiliser des formats autres : OpenAPI (ex Swagger) ou RAML

# Plan du cours

- Généralités JAX-RS
- Premier service web JAX-RS
- Développement serveur
  - Ressources : *@Path*
  - Méthodes : *@POST*, *@PUT*, *@DELETE* et *@GET*
  - Représentation : gestion du contenu et *Response*
- Développement client et test d'intégration
- Déploiement



# Le premier service web JAX-RS

## ➤ Exemple : service web REST « HelloWorld »

Définition d'un chemin de ressource pour associer une ressource *hello* à une URI

```
@Path("/hello")
public class HelloWorldResource {

    @GET
    @Produces("text/plain")
    public String getHelloWorld() {
        return "Hello World from text/plain";
    }
}
```

Lecture de la ressource *HelloWorld* via une requête HTTP de type GET

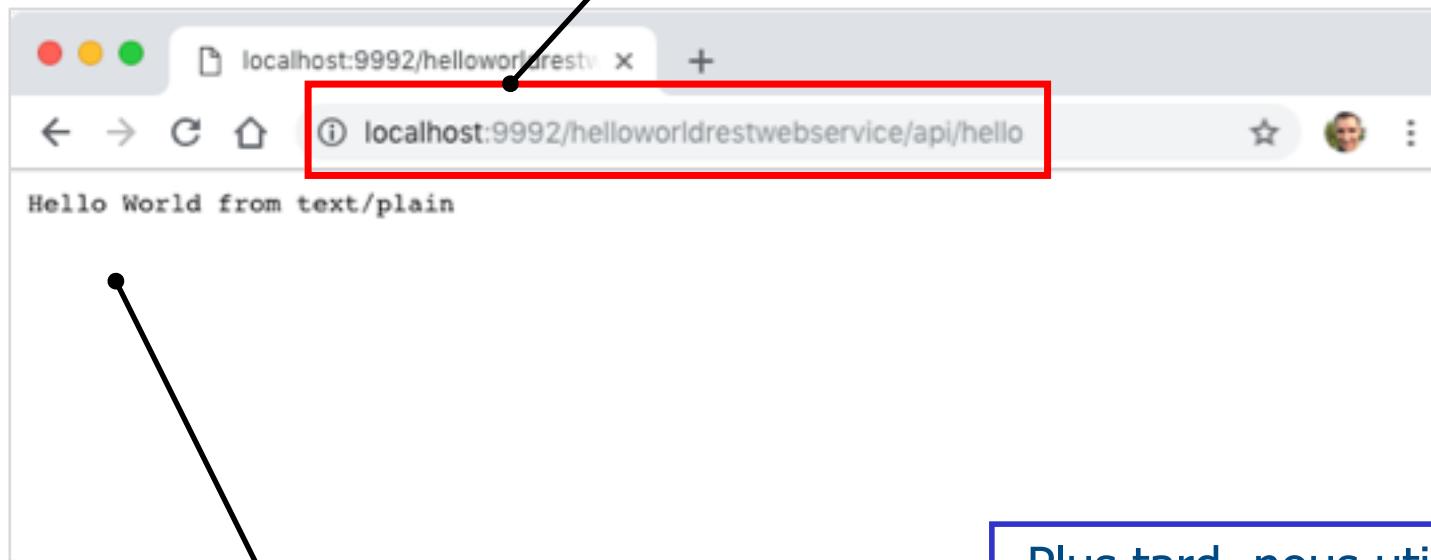
Le type MIME de la réponse est de type *text/plain*

*HelloWorldResource.java* du projet **jaxrs-helloworldrestwebservice**

# Le premier service web JAX-RS

## ➤ Exemple (suite) : service web REST « HelloWorld »

Envoi d'une requête HTTP de type GET  
demandant la lecture de la ressource  
*hello*



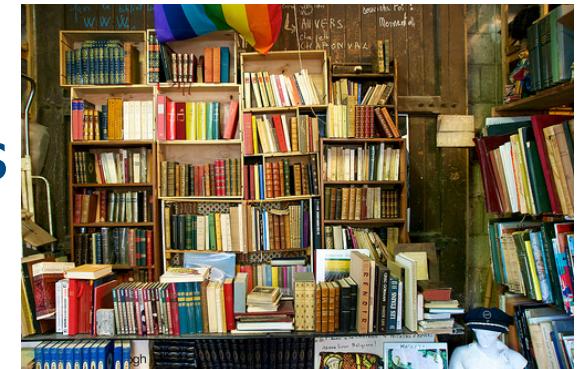
Le retour est directement interprétable  
depuis la navigateur puisqu'il s'agit d'un  
type MIME reconnu

Plus tard, nous utiliserons des  
outils qui facilitent l'écriture de  
requêtes HTTP plus complexes  
(POST, PUT...)



## Exemple file rouge : une bibliothèque

- Utilisation d'un exemple représentatif pour la présentation des concepts de JAX-RS : une **Bibliothèque**
- Mise en place d'un système de **CRUD**
- **Bibliothèque** et **livre** sont des ressources
- Description de l'exemple
  - Une bibliothèque dispose de livres
  - Possibilité d'ajouter, de mettre à jour ou de supprimer un livre
  - Recherche d'un livre en fonction de différents critères (ISBN, nom...)
  - Récupération de données de types simples (String, Long...) ou de type objet
  - Différents formats de données (JSON, XML...)



# Plan du cours

- Généralités JAX-RS
- Premier service web JAX-RS
- Développement serveur
  - Ressources : *@Path*
  - Méthodes : *@POST*, *@PUT*, *@DELETE* et *@GET*
  - Représentation : gestion du contenu et *Response*
- Développement client et test d'intégration
- Déploiement



# C'est quoi REST ? Avec JAX-RS

## ➤ Ressources (Identifiant)

- Identifié par une URI
- Exemple : *http://localhost:8080/libraryrestwebservice/api/books*
- *@Path, @PathParam, @QueryParam, @FormParam, @HeaderParam*

## ➤ Méthodes (Verbes) pour manipuler l'identifiant

- Méthodes HTTP : GET, POST, PUT and DELETE
- *@GET, @POST, @PUT and @DELETE*

## ➤ Représentation donne une vue sur l'état

- Informations transférées entre le client et le serveur
- Exemples : XML, JSON...
- *@Produces et @Consumes*

## @Path

- Une classe Java doit être annotée par *@Path* pour qu'elle puisse être traitée par des requêtes HTTP
- L'annotation *@Path* sur une classe définit des ressources appelées racines (**Root Resource Class**)
- La valeur donnée à *@Path* correspond à une expression URI relative au contexte de l'application web

*http://localhost:8088/libraryrestwebservice/api/books*

Adresse du Serveur

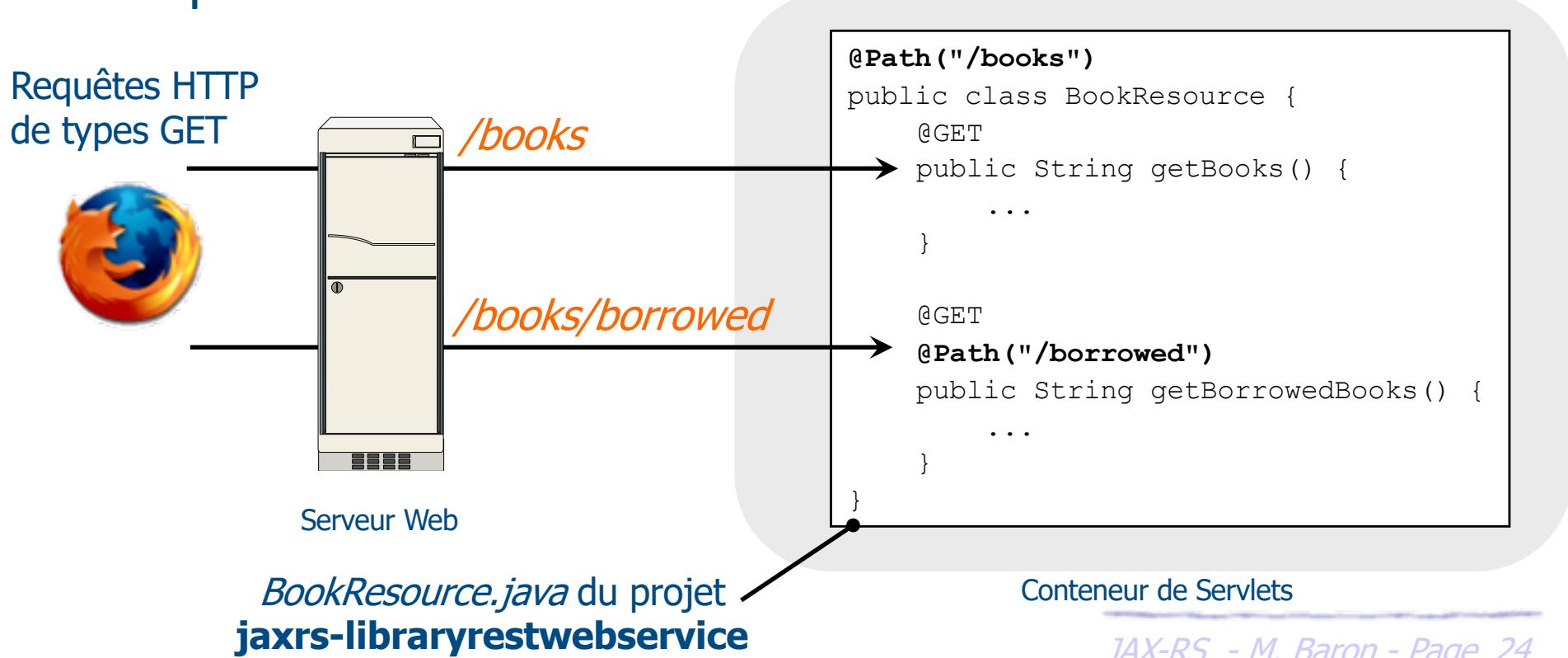
Port

Contexte de l'application WEB

URI de la ressource

## @Path

- L'annotation *@Path* peut également annoter des méthodes de la classe (facultatif)
- L'URI résultante est la concaténation de l'expression du *@Path* de la classe avec l'expression du *@Path* de la méthode
- Exemple



## @Path : Template Parameters

- La valeur définie dans *@Path* ne se limite pas seulement aux expressions constantes
- Possibilité de définir des expressions plus complexes appelées **Template Parameters**
- Pour distinguer une expression complexe dans la valeur du *@Path*, son contenu est délimité par { ... }
- Possibilité également de mixer dans la valeur de *@Path* des expressions constantes et des expressions complexes
- Les **Template Parameters** peuvent également utiliser des expressions régulières

# @Path : Template Parameters

- Exemple : récupérer un livre par son identifiant

```
@Path("/books")
public class BookResource {
    ...

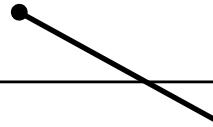
    @GET
    @Path("{id}")
    public String getBookById(@PathParam("id") int id) {
        return "Java For Life " + id;
    }

    @GET
    @Path("name-{name}-editor-{editor}")
    public String getBookByNameAndEditor(@PathParam("name") String name,
                                         @PathParam("editor") String editor)
        return "Starcraft 2 for Dummies (Name:" + name + " - Editor:" + editor + ")";
    }
}
```

/books/123



/books/name-sc2-editor-oreilly



*BookResource.java du projet  
jaxrs-libraryrestwebservice*



# @Path : Template Parameters

- Exemple (bis) : récupérer un livre par son identifiant

*/books/123/path1/path2/editor*

```
@Path("/books")
public class BookResource {
    ...
    @GET
    @Path("{id : .+}/editor")
    public String getBookEditorById(@PathParam("id") String id) {
        return "Moira";
    }

    @GET
    @Path("original/{id : .+}")
    public String getOriginalBookById(@PathParam("id") String id) {
        return "Science will reveal the truth";
    }
}
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*

*/books/original/123/path1/path2*

## @Path : sub-resource locator

- Une **sub-resource locator** est une méthode qui doit respecter les exigences suivantes
  - Annotée avec `@Path`
  - Non annotée avec `@GET`, `@POST`, `@PUT`, `@DELETE`
  - Retourne une sous ressource (un type *Object*)
- L'intérêt d'utiliser une méthode **sub-resource locator** est de pouvoir déléguer vers une autre classe ressource
- Le développement d'une sous ressource suit un schéma classique, pas d'obligation de placer une ressource racine
- Une méthode **sub-resource locator** supporte le polymorphisme (retourne des sous types)

# @Path : sub-resource locator

- Exemple : appeler une méthode **Sub-resource locator**

*BookResource.java du projet  
jaxrs-libraryrestwebservice*

```
@Path("/books")
public class BookResource {
    ...
    @Path("specific")
    public SpecificBookResource getSpecificBook() {
        return new SpecificBookResource();
    }
}
```

Méthode **Sub-resource locator** dont la sous ressource est définie par *SpecificBookResource*

```
public class SpecificBookResource {
    @GET
    @Path("{id}")
    public String getSpecificBookById(@PathParam("id") int id) {
        return ".NET platform is Bad";
    }
}
```

*SpecificBookResource.java du projet  
jaxrs-libraryrestwebservice*

*/books/specific/123*



# Plan du cours

- Généralités JAX-RS

- Premier service web JAX-RS

- Développement serveur

- Ressources : *@Path*

- Méthodes : ***@POST, @PUT, @DELETE et @GET***

- Représentation : gestion du contenu et *Response*

- Développement client et test d'intégration

- Déploiement

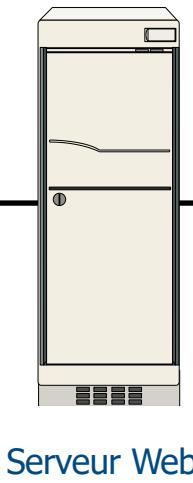


# @GET, @POST, @PUT, @DELETE : méthodes HTTP

- La spécification JAX-RS, n'impose pas de respecter les conventions définies par le style REST
  - Possibilité d'utiliser une requête HTTP de type GET pour effectuer une suppression d'une ressource (**Attention au non sens**)
- Des opérations CRUD sur des ressources sont réalisées au travers des méthodes HTTP



Requêtes HTTP  
GET,  
POST,  
PUT et  
DELETE



Serveur Web

*/books*  
**GET** : récupère la liste de tous les livres  
**POST** : créer un nouveau livre

*/books/{id}*  
**GET** : récupère un livre  
**PUT** : mise à jour d'un livre  
**DELETE** : effacer un livre

Conteneur de Servlets

## @GET, @POST, @PUT, @DELETE : méthodes HTTP

- L'annotation des méthodes Java permet de traiter de requêtes HTTP suivant le type de méthode (GET, POST...)
- Les annotations disponibles par JAX-RS sont les suivantes
  - @GET, @POST, @PUT, @DELETE et @HEAD
- Ces annotations ne sont utilisables que sur des méthodes Java
- Le nom des méthodes Java n'a pas d'importance puisque c'est l'annotation employée qui précise où se fera le traitement
- Possibilité d'étendre les annotations disponibles pour gérer différents types de méthode HTTP
  - Protocole WebDav (extension au protocole HTTP pour la gestion de documents)
  - Méthodes supportées : PROPFIND, COPY, MOVE, LOCK, UNLOCK...

# @GET, @POST, @PUT, @DELETE : méthodes HTTP

## ➤ Exemple : CRUD sur la ressource Livre

```
@Path("/books")
public class BookResource {
    @GET
    public String getBooks() {
        return "Cuisine et moi / Java 18";
    }
    @POST
    public String createBook(String livre) {
        return livre;
    }
    @GET
    @Path("{id}")
    public String getBookById(@PathParam("id") int id) {
        return "Java For Life " + id;
    }
    @PUT
    @Path("{id}")
    public void updateBookById(@PathParam("id") int id, String livre) {
        ...
    }
    @DELETE
    @Path("{id}")
    public void deleteBookById(@PathParam("id") int id) {
        ...
    }
}
```

Récupérer la liste de tous les livres  
Créer un nouveau livre  
Récupérer un livre  
Mettre à jour un livre  
Effacer un livre

*BookResource.java du projet  
jaxrs-libraryrestwebservice*



# Paramètres de requêtes

- JAX-RS fournit des annotations pour extraire des paramètres d'une requête
- Elles sont utilisées sur les paramètres des méthodes des ressources pour réaliser l'injection du contenu
- Liste des différentes annotations disponibles
  - *@PathParam* : extraire les valeurs des **Template Parameters**
  - *@QueryParam* : extraire les valeurs des paramètres de requête
  - *@FormParam* : extraire les valeurs des paramètres de formulaire
  - *@HeaderParam* : extraire les paramètres de l'en-tête
  - *@CookieParam* : extraire les paramètres des cookies
  - *@Context* : extraire les informations liées aux ressources de contexte

# Paramètres de requêtes : fonctionnalités communes

- Une valeur par défaut peut être spécifiée en utilisant l'annotation *@DefaultValue*
- Par défaut, JAX-RS décode tous les paramètres, la résolution de l'encodage se fait par l'annotation *@Encoded*
- Les annotations peuvent être utilisées sur les types Java suivants
  - Les types primitifs sauf *char* et les classes qui les encapsulent
  - Toutes classes ayant un constructeur avec paramètre de type *String*
  - Toutes classes ayant la méthode statique *valueOf(String)*
  - *List<T>*, *Set<T>* et *SortedSet<T>*

# Paramètres de requêtes : @PathParam

- L'annotation `@PathParam` est utilisée pour extraire les valeurs des paramètres contenues dans les **Template Parameters**
- Exemple

```
@Path("/books")
public class BookResource {
    ...
    @GET
    @Path("{id}")
    public String getBookById(@PathParam("id") int id) {
        return "Java For Life " + id;
    }
    ...
    @GET
    @Path("name-{name}-editor-{editor}")
    public String getBookByNameAndEditor(@PathParam("name") String name,
                                         @PathParam("editor") String editor)
        return "Starcraft 2 for Dummies (Name:" + name + " - Editor:" + editor + ")";
}
```

The diagram illustrates the effect of annotations on the code. It shows two examples of template parameters being mapped to method parameters:

- A callout points from the annotation `@PathParam("id")` in the first method to the URL fragment `/books/123/path1/path2/editor`. A text box next to it states: "Injecte les valeurs dans les paramètres de la méthode".
- A callout points from the annotations `@PathParam("name")` and `@PathParam("editor")` in the second method to the URL fragment `/books/name-sc2-editor-oreilly`.
- A callout points from the entire code block to the text `BookResource.java du projet jaxrs-libraryrestwebservice`.



# Paramètres de requêtes : `@QueryParam`

- L'annotation `@QueryParam` est utilisée pour extraire les valeurs des paramètres contenues dans une requête quelque soit son type de méthode HTTP
- Exemple

*/books/queryparameters?name=harry&isbn=1-111111-11&isExtended=true*

```
@Path("/books")
public class BookResource {
    ...
    @GET
    @Path("queryparameters")
    public String getQueryParameterBook(
        @DefaultValue("all") @QueryParam("name") String name,
        @DefaultValue("-??????-?") @QueryParam("isbn") String isbn,
        @DefaultValue("false") @QueryParam("isExtended") boolean isExtented) {

        return name + " " + isbn + " " + isExtented;
    }
}
```

Injection de valeurs par défaut si les valeurs des paramètres ne sont pas fournies

*BookResource.java du projet  
jaxrs-libraryrestwebservice*

JAX-RS - M. Baron - Page 37



# Paramètres de requêtes : `@FormParam`

- L'annotation `@FormParam` est utilisée pour extraire les valeurs des paramètres contenues dans un formulaire
- Le type de contenu doit être `application/x-www-form-urlencoded`
- Cette annotation est très utile pour extraire les informations d'une requête POST d'un formulaire HTML
- Exemple

```
@Path("/books")
public class BookResource {
    ...
    @POST
    @Path("createfromform")
    @Consumes("application/x-www-form-urlencoded")
    public String createBookFromForm(@FormParam("name") String name) {
        System.out.println("BookResource.createBookFromForm()");
        return name;
    }
}
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*

# Paramètres de requêtes : `@HeaderParam`

- L'annotation `@HeaderParam` est utilisée pour extraire les paramètres contenues dans l'en-tête d'une requête
- Exemple

```
@Path("/books")
public class BookResource {
    ...
    @GET
    @Path("headerparameters")
    public String getHeaderParameterBook(
        @DefaultValue("all") @HeaderParam("name") String name,
        @DefaultValue("-?-?????-?") @HeaderParam("isbn") String isbn,
        @DefaultValue("false") @HeaderParam("isExtended") boolean isExtented) {
        return name + " " + isbn + " " + isExtented;
    }
}
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*



## Paramètres de requêtes : @Context

- L'annotation *@Context* permet d'injecter des objets liés au contexte de l'application
- Les types d'objets supportés sont les suivants
  - *UriInfo* : informations liées aux URIs
  - *Request* : informations liées au traitement de la requête
  - *HttpHeaders* : informations liées à l'en-tête
  - *SecurityContext* : informations liées à la sécurité
- Certains de ces objets permettent d'obtenir les mêmes informations que les précédentes annotations liées aux paramètres

# Paramètres de requêtes : @Context / UriInfo

- Un objet de type *UriInfo* permet d'extraire les informations « brutes » d'une requête HTTP
- Les principales méthodes sont les suivantes
  - *String getPath()* : chemin relatif de la requête
  - *MultivaluedMap<String, String> getPathParameters()* : valeurs des paramètres de la requête contenues dans **Template Parameters**
  - *MultivaluedMap<String, String> getQueryParameters()* : valeurs des paramètres de la requête
  - *URI getBaseUri()* : chemin de l'application
  - *URI getAbsolutePath()* : chemin absolu (base + chemins)
  - *URI getRequestUri()* : chemin absolu incluant les paramètres

# Paramètres de requêtes : @Context / UriInfo

- Exemple : accéder aux informations d'une requête via *UriInfo*

*/books/informationfromuriinfo/test?param1=value&param2=value*

```
@Path("/books")
public class BookResource {
    ...
    @GET
    @Path("informationfromuriinfo/{name}")
    public String getInformationFromUriInfo(@Context UriInfo uriInfo, @PathParam("name") String name) {
        result.append("getPath(): " + uriInfo.getPath() + "\n");
        List<PathSegment> pathSegments = uriInfo.getPathSegments();
        ...
        MultivaluedMap<String, String> pathParameters = uriInfo.getPathParameters();
        ...
        MultivaluedMap<String, String> queryParameters = uriInfo.getQueryParameters();
        ...
        result.append("getAbsolutePath(): " + uriInfo.getAbsolutePath() + "\n");
        result.append("getBaseUri(): " + uriInfo.getBaseUri() + "\n");
        result.append("getRequestUri(): " + uriInfo.getRequestUri() + "\n");
        return ...;
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*



The screenshot shows an IDE interface with the BookResource.java code in the editor. Below it, the Console tab displays the output of the application's execution. The output shows the results of various UriInfo methods being called on the URL '/books/informationfromuriinfo/test'. The methods listed are getPath(), getPathSegments(), getPathParameters(), getQueryParameters(), getAbsolutePath(), getBaseUri(), and getRequestUri(). The output also includes the parameter values param1 and param2.

```
Problems Javadoc Declaration Search Console
LibraryRestWebServiceLauncher [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home/bin/java (Oct 18, 2018, 5:17:11 PM)
getPath(): books/informationfromuriinfo/test
getPathSegments(): books informationfromuriinfo test
getPathParameters(): name
getQueryParameters(): param1 param2
getAbsolutePath(): http://localhost:9992/libraryrestwebservice/api/books/informationfromuriinfo/test
getBaseUri(): http://localhost:9992/libraryrestwebservice/api/
getRequestUri(): http://localhost:9992/libraryrestwebservice/api/books/informationfromuriinfo/test?param1=value&param2=value
```

## Paramètres de requêtes : @Context / HttpHeaders

- Un objet de type *HttpHeader* permet d'extraire les informations contenues dans l'en-tête d'une requête
- Les principales méthodes sont les suivantes
  - *Map<String, Cookie> getCookies()* : les cookies de la requête
  - *Locale getLanguage()* : la langue de la requête
  - *MultivaluedMap<String, String> getRequestHeaders()* : valeurs des paramètres de l'en-tête de la requête
  - *MediaType getMediaType()* : le type MIME de la requête
- À noter que ces méthodes permettent d'obtenir le même résultat que les annotations *@HeaderParam* et *@CookieParam*

# Paramètres de requêtes : @Context / HttpHeaders

## ➤ Exemple : accéder aux informations à l'en-tête

```
@Path("/books")
public class BookResource {
    ...
    @GET
    @Path("informationfromhttpheaders")
    public String getInformationFromHttpHeaders(@Context HttpHeaders httpheaders) {
        Map<String, Cookie> cookies = httpheaders.get Cookies();
        Set<String> currentKeySet = cookies.keySet();
        for (String currentCookie : currentKeySet) {
            result.append(currentCookie + "\n");
        }

        MultivaluedMap<String, String> requestHeaders = httpheaders.getRequestHeaders();
        Set<String> requestHeadersSet = requestHeaders.keySet();
        for (String currentHeader : requestHeadersSet) {
            result.append(currentHeader + ": " + requestHeaders.get(currentHeader) + "\n");
        }
    }
}
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*



The screenshot shows an IDE interface with a toolbar at the top and a code editor below. The code editor contains the Java code for BookResource.java. In the bottom right corner of the code editor, the URL */books/informationfromhttpheaders* is displayed in orange. Below the code editor, the IDE's console window shows the output of the application. The output lists various HTTP headers sent by the browser, such as host, connection, cache-control, upgrade-insecure-requests, user-agent, accept, accept-encoding, accept-language, and cookie.

```
Problems Javadoc Declaration Search Console
LibraryRestWebServiceLauncher [Java Application] /Library/Java/JavaVirtualMachines/jdk-11.jdk/Contents/Home/bin/java (Oct 18, 2018, 5:38:06 PM)
Cookies:
_ga
RequestHeaders:
host: [localhost:9992]
connection: [keep-alive]
cache-control: [max-age=0]
upgrade-insecure-requests: [1]
user-agent: [Mozilla/5.0 (Macintosh; Intel Mac OS X 10_14_0) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/69.0.3497.100 Safari/537.36]
accept: [text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,*/*;q=0.8]
accept-encoding: [gzip, deflate, br]
accept-language: [fr-FR,fr;q=0.9,en-US;q=0.8,en;q=0.7,la;q=0.6,ca;q=0.5]
cookie: [_ga=GA1.1.1348631373.1498588093]
```



# Plan du cours

- Généralités JAX-RS

- Premier service web JAX-RS

- Développement serveur

- Ressources : *@Path*

- Méthodes : *@POST*, *@PUT*, *@DELETE* et *@GET*

- **Représentation : gestion du contenu et *Response***

- Développement client et test d'intégration

- Déploiement

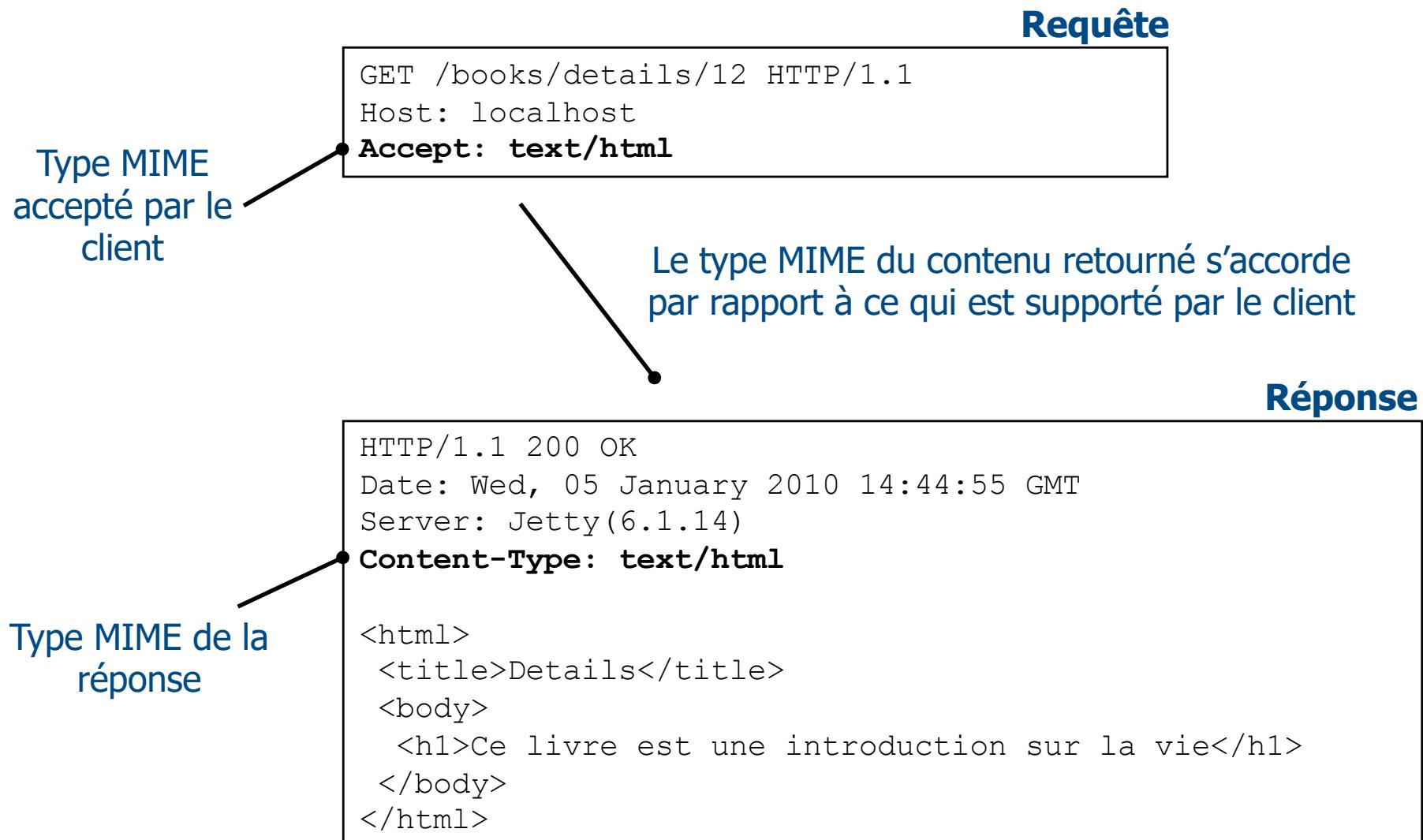


## Représentations : @Consumes et @Produces

- L'annotation *@Consumes* est utilisée pour spécifier le ou les types MIME qu'une méthode d'une ressource peut accepter
- L'annotation *@Produces* est utilisée pour spécifier le ou les types MIME qu'une méthode d'une ressource peut produire
- Possibilité de définir un ou plusieurs types MIME
- Ces annotations peuvent porter sur : classe ou méthode
  - L'annotation sur la méthode surcharge celle de la classe
- Si ces annotations ne sont pas utilisées tous types MIME pourront être acceptés ou produits
- La liste des constantes des différents types MIME est disponible dans la classe *MediaType*

# Représentations : @Consumes et @Produces

## ➤ Exemple : gestion du type MIME



# Représentations : @Consumes et @Produces

## ➤ Exemple (suite) : gestion du type MIME

```
@Path("/books")
public class BookResource {
    ...
    @GET
    @Path("details/{id}")
    @Produces(MediaType.TEXT_PLAIN)
    public String getDetailTextBookId(@PathParam("id") String id) {
        return "Ce livre est une introduction sur la vie";
    }
    @GET
    @Path("details/{id}")
    @Produces(MediaType.TEXT_XML)
    public String getDetailXMLBookId(@PathParam("id") String id) {
        return "<?xml version=\"1.0\"?>" + "<details>Ce livre est une introduction sur la
vie" + "</details>";
    }
    @GET
    @Path("details/{id}")
    @Produces(MediaType.TEXT_HTML)
    public String getDetailHTMLBookId(@PathParam("id") String id) {
        return "<html> " + "<title>" + "Details" + "</title>" + "<body><h1>" + "Ce livre
est une introduction sur la vie" + "</body></h1>" + "</html> ";
    }
}
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*

Le chemin des trois méthodes est identique.  
L'ambiguïté est levé par le type de contenu supporté

Le choix de la méthode déclenchée dépend  
du type MIME supporté par le client et de  
la priorité accordée



## Gestion du contenu : généralités

- Les points précédents se sont focalisés sur les informations contenues dans l'en-tête d'une requête
- JAX-RS permet également de manipuler le contenu du corps d'une requête et d'une réponse
  - Un livre est un objet (pas qu'un *String*)
  - Un livre peut référencer du contenu binaire (fichier PDF, image...)
- JAX-RS peut automatiquement effectuer des opérations de sérialisation et dé-sérialisation vers un type Java spécifique
  - */\* : byte[]*
  - *text/\* : String*
  - *text/xml, application/xml, application/\*+xml : JAXBElement*
  - *application/x-www-form-urlencoded : Multimap<String, String>*

# Gestion du contenu : *InputStream*

## ➤ Exemple : requête et réponse avec un flux d'entrée

```
@Path("/contentbooks")
public class BookContentResource {
    @PUT
    @Path("inputstream")
    public void updateContentBookWithInputStream(InputStream is) throws IOException {
        byte[] bytes = readFromStream(is);
        return new String(bytes);
    }

    private byte[] readFromStream(InputStream stream) throws IOException {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        int wasRead = 0;
        byte[] buffer = new byte[1024];
        while ((wasRead = stream.read(buffer, 0, buffer.length)) != -1) {
            baos.write(buffer, 0, wasRead);
        }
        return baos.toByteArray();
    }

    @GET
    @Path("inputstream")
    @Produces(MediaType.TEXT_PLAIN)
    public InputStream getContentBookWithInputStream() throws FileNotFoundException {
        return new FileInputStream("src/main/resources/sample.txt");
    }
}
```

*BookContentResource.java* du projet  
**jaxrs-libraryrestwebservice**



# Gestion du contenu : *File*

## ➤ Exemple : requête et réponse avec un fichier

```
@Path("/contentbooks")
public class BookContentResource {
    @Path("file")
    @PUT
    public void updateContentBookWithFile(File file) throws IOException {
        byte[] bytes = readFromStream(new FileInputStream(file));
        String input = new String(bytes);
        System.out.println(input);
    }

    @Path("file")
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public File getContentBookWithFile() {
        File file = new File("src/main/resources/sample.txt");
        return file;
    }

    ...
}
```

JAX-RS crée un fichier temporaire  
à partir du fichier donné

*BookContentResource.java* du projet  
**jaxrs-libraryrestwebservice**



## Gestion du contenu : types personnalisés en XML

- JAX-RS offre la possibilité d'utiliser des types personnalisés en s'appuyant sur la spécification **JAXB** (JSR 222)
- C'est une spécification qui permet de mapper des classes Java en XML et en XML Schema
- L'avantage est de pouvoir manipuler directement des objets Java sans passer par une représentation abstraite XML
- Chaque classe **doit être annotée à la racine** pour activer le mapping entre l'XML Schema et les attributs de la classe
  - ***XmRootElement, XmElement, XmType...***



# Gestion du contenu : types personnalisés en XML

## ➤ Exemple : mise à jour d'un livre (format XML)

```
@XmlRootElement(name = "book")
public class Book {
    protected String name;

    protected String isbn;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }

    public String toString() {
        return name;
    }
}
```

Annotation JAXB **obligatoire**  
pour définir l'élément racine  
de l'arbre XML

*Book.java* du projet  
**jaxrs-libraryrestwebservice**



# Gestion du contenu : types personnalisés en XML

## ➤ Exemple (suite) : mise à jour d'un livre (format XML)

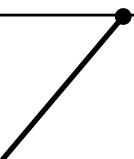
```
@Path("/contentbooks")
public class BookContentResource {
    @Path("xml")
    @Consumes(MediaType.APPLICATION_XML)
    @PUT
    public void updateContentBookWithXML(Book current) throws IOException {
        System.out.println("Name: " + current.getName() + ", ISBN: " + current.getIsbn());
    }

    @Path("xml")
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public Book getContentBookWithXML() {
        Book current = new Book();
        current.setIsbn("1-111111-11");
        current.setName("Harry");

        return current;
    }

    ...
}
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*



# Gestion du contenu : types personnalisés en XML

## ➤ Exemple (suite) : dépendances Maven obligatoires

```
...
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>com.sun.istack</groupId>
    <artifactId>istack-commons-runtime</artifactId>
    <version>3.0.7</version>
</dependency>
...
```

*pom.xml du projet  
**jaxrs-libraryrestwebservice***



## Gestion du contenu : types personnalisés en JSON

- JAX-RS offre aussi la possibilité d'utiliser la sérialisation d'objet vers le format JSON et inversement (Binding)
- L'avantage est de pouvoir manipuler directement des objets Java sans passer par une représentation JSON
- Contrairement à JAXB pas besoin d'annoter les classes à la racine pour rendre actif le mapping
- Annotations disponibles
  - `@JsonGetter, @JsonPropertyOrder...`
  - <https://www.baeldung.com/jackson-annotations>



# Gestion du contenu : types personnalisés en JSON

## ➤ Exemple : mise à jour d'un livre (format JSON)

```
@JsonPropertyOrder({ "name", "isbn" })
public class Book {
    @JsonProperty("book_name")
    protected String name;

    @JsonProperty("book_isbn")
    protected String isbn;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getIsbn() {
        return isbn;
    }

    public void setIsbn(String isbn) {
        this.isbn = isbn;
    }
    ...
}
```

*Book.java du projet  
jaxrs-libraryrestwebservice*

```
@XmlRootElement(name = "book")
@JsonPropertyOrder({ "name", "isbn" })
public class Book {
    @JsonProperty("book_name")
    protected String name;

    @JsonProperty("book_isbn")
    protected String isbn;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getIsbn() {
        return isbn;
    }

    ...
}
```

Possibilité de combiner les annotations Jackson et JAXB



# Gestion du contenu : types personnalisés en JSON

## ➤ Exemple (suite) : mise à jour d'un livre (format JSON)

```
@Path("/contentbooks")
public class BookContentResource {
    ...

    @Path("json")
    @Consumes(MediaType.APPLICATION_JSON)
    @PUT
    public void updateContentBookWithJSON(Book current) {
        ...
    }

    @Path("json")
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public Book getContentBookWithJSON() {
        Book current = new Book();
        current.setIsbn("1-111111-11");
        current.setName("Harry");

        return current;
    }

    ...
}
```

*BookResource.java du projet  
jaxrs-libraryrestwebservice*



# Gestion du contenu : types personnalisés en JSON

- Jersey s'appuie sur des implémentations pour fournir le mécanisme de binding Java / JSON
- Différentes implémentations disponibles
  - *Jackson, JSON-B, MOXy...*
- Exemple (suite) : mise à jour d'un livre (format JSON)

```
...
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-jackson</artifactId>
</dependency>
...
```

*pom.xml* du projet  
**jaxrs-libraryrestwebservice**

## Response : objectif

- Actuellement, tous les services développés retournaient *void*, *String* ou un type personnalisé Java
- Comment retourner plusieurs types d'information ?
  - Un code de retour
  - Des paramètres pour l'en-tête
  - Un objet
  - Le format de l'objet (text/plain, JSON...)
  - Une URI...
- Utilisation de la classe *Response* pour « transporter » plusieurs informations
- Un objet *Response* est construit à partir d'une API reposant sur le patron de conception **Builder**

```
Response.status(Response.Status.OK).header("param1", "value1").entity("Message").build();
```

# Response : construction (Builder)

## 1) Construction du **statut**

- *status(Status s)* : définit un statut depuis *Response.Status*
- ou
- *accepted(), ok(), serverError()...* : méthodes prêtes à l'emploi avec un statut imposée

## 2) Construction de l'**en-tête** et du **corps**

- *header(String, Object)* : ajoute un paramètre à l'en-tête
- *entity(Object)* : renseigne le contenu du corps
- *type(MediaType)* : précise le type

L'annotation *@Produces* fournit le même résultat que d'utiliser *type(MediaType)*



## 3) Construction **finale**

- *Response build()* : méthode terminale

# Response : statut

## ➤ Réponse sans erreur

- Les statuts des réponses sans erreur s'échelonnent de 200 à 399
- Le code est 200 « OK » pour les retours de contenus non vides
- Le code est 204 « No Content » pour les services retournant un contenu vide
  - *void* ou valeur *null* d'un type personnalisé Java

## ➤ Réponse avec erreur

- Les statuts des réponses avec erreur s'échelonnent de 400 à 599
- Une ressource non trouvée, le code de retour est 404 « Not Found »
- Un type MIME en retour non supporté, 406 « Not Acceptable »
- Une méthode HTTP non supportée, 405 « Method Not Allowed »
- Des paramètres manquants, 400 « Bad Request »
- Une ressource non modifiée, 304 « Not Modified »
- Un gros problème, 500 « Internal Server Error »

# Response

## ➤ Exemple 1 : statut + objet dans un objet *Response*

```
@Path("/responsebooks")
public class BookResponseResource {
    ...
    @GET
    @Path("ok/without_response")
    public String getBookWithoutResponse() {
        return "Java For Life";
    }

    @GET
    @Path("ok")
    public Response getBook() {
        return Response.status(Response.Status.OK).entity("Java For Life").build();
    }
}
```

Exceptés les chemins, les réponses retournées fournissent le même résultat

*BookResponseResource.java* du projet  
**jaxrs-libraryrestwebservice**

# Response

## ➤ Exemple 2 : paramètres d'en-tête dans un objet *Response*

*BookResponseResource.java* du projet  
**jaxrs-libraryrestwebservice**

```
@Path("/responsebooks")
public class BookResponseResource {
    ...
    @GET
    @Path("ok/headers")
    public String getBookWithHeaders() {
        return Response.status(Response.Status.OK).entity("Java For Life").header("param1", "value1").build();
    }
}
```

L'objet *Response* permettra de transmettre des paramètres d'en-tête

```
$ curl -v http://localhost:9992/libraryrestwebservice/api/responsebooks/ok/headers
* Connected to localhost (127.0.0.1) port 9992 (#0)
> GET /libraryrestwebservice/api/responsebooks/ok/headers HTTP/1.1
> Host: localhost:9992
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 200 OK
< param1: value1
< Content-Type: text/plain
< Content-Length: 13
<
* Connection #0 to host localhost left intact
Java For Life
```



# Response

## ➤ Exemple 3 : objet et type personnalisés dans un *Response*

```
@Path("/responsebooks")
public class BookResponseResource {
    ...
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("ok/json_annotation")
    public Book getBookJSONAnnotation() {
        Book current = new Book();
        current.setIsbn("1-111111-11");
        current.setName("Harry");
        return current
    }

    @GET
    @Path("ok/json")
    public Response getBookJSON() {
        Book current = new Book();
        current.setIsbn("1-111111-11");
        current.setName("Harry");
        return Response.status(Response.Status.OK).entity(current).type(MediaType.APPLICATION_JSON).build();
    }
}
```

*BookResponseResource.java* du projet  
**jaxrs-libraryrestwebservice**

Possibilité de choisir  
« programmatiquement » le  
type de la réponse

Mise à part les chemins, les  
réponses retournées  
fournissent le même résultat

## Response avec erreur

- Les codes d'erreur permettent de préciser au client que des problèmes sont survenus sur le serveur
- Deux façons de transmettre les codes d'erreurs
  - Via *Response* en précisant le code statut dans *status(Status s)*
  - En levant une exception *WebApplicationException* ou un sous type
    - *BadRequestException*, *NotFoundException*...
- Lever *WebApplicationException*, quelles différences ?
  - Ne pas retourner explicitement un objet *Response*
  - Encapsule implicitement un objet *Response* (transparent pour le développeur)
  - Traiter l'erreur comme une exception (trace complète)

# Response avec erreur

## ➤ Exemple : retourner une réponse avec un statut erreur

```
@Path("/responsebooks")
public class BookResponseResource {
    ...
    @GET
    @Path("error/webapplicationexception")
    public String getBookWithWebApplicationException(@QueryParam("id") String id) {
        if (null == id) {
            throw new BadRequestException();
        }
        return "Java For Life" + id;
    }

    @GET
    @Path("error")
    public Response getBookWithError(@QueryParam("id") String id) {
        if (null == id) {
            return Response.status(Response.Status.BAD_REQUEST).build();
        }
        return Response.status(Response.Status.OK).entity("Java For Life" + id).build();
    }
}
```

*BookResponseResource.java du projet  
jaxrs-libraryrestbservice*

Mise à part les chemins, les réponses retournées fournissent le même résultat

```
$ curl -v http://localhost:9992/libraryrestbservice/api/responsebooks/error
* Connected to localhost (127.0.0.1) port 9992 (#0)
> GET /libraryrestbservice/api/responsebooks/error HTTP/1.1
> Host: localhost:9992
> User-Agent: curl/7.54.0
> Accept: */*
< HTTP/1.1 400 Bad Request
```



# Response : pour ou contre ?

## ➤ Pour *Response*

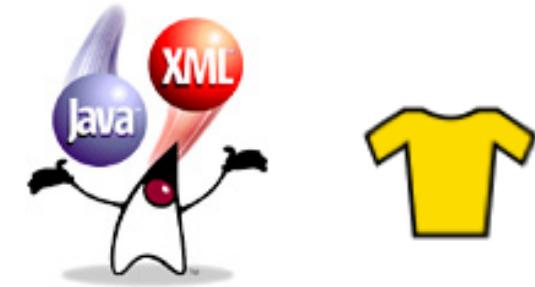
- Fournir « programmatiquement » le type de la réponse
- Fournir des statuts différents à 200 (si ce ne sont pas des erreurs)
- Ajouter des en-têtes à la réponse

## ➤ Contre *Response*

- Implique que pour les tests unitaires une dépendance vers JAX-RS soit nécessaire (framework dépendant)
- Si erreur utilisation d'une exception de type *WebApplicationException*
- Pour le type de la réponse utilisation de la notation *@Produces*
- Pour résumer ...
- Si possible, éviter d'utiliser *Response*

# Plan du cours

- Généralités JAX-RS
- Premier service web JAX-RS
- Développement serveur
  - Ressources : *@Path*
  - Méthodes : *@POST*, *@PUT*, *@DELETE* et *@GET*
  - Représentation : gestion du contenu et *Response*
- **Développement client et test d'intégration**
- Déploiement



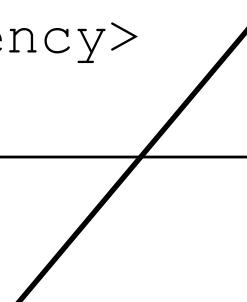
## Développement client

- La spécification JAX-RS depuis 2.0 s'intéresse à fournir une API pour le traitement côté client
- Possibilité également d'utiliser des bibliothèques spécialisées dans l'envoi et la réception de requêtes HTTP
- L'utilisation de l'API cliente ne suppose pas que les services web soient développés avec JAX-RS (.NET, PHP...)
- Les avantages d'utiliser l'API cliente de JERSEY
  - Manipuler les types Java (pas de transformation explicite en XML)
  - Facilite l'écriture des tests d'intégration

# Développement client

## ➤ Ajout de dépendance Maven

```
...
<dependency>
    <groupId>org.glassfish.jersey.core</groupId>
    <artifactId>jersey-client</artifactId>
</dependency>
...
```



Seule cette dépendance est nécessaire pour le support de l'API cliente de JAX-RS depuis Jersey

# Développement client : 1 – initialiser un client

## ➤ Création d'un client

```
Client client = ClientBuilder.newClient();
```

## ➤ Création d'un client avec une configuration

```
ClientConfig clientConfig = new ClientConfig();
clientConfig...
Client client = ClientBuilder.newClient(clientConfig);
```

## ➤ Que préciser dans une configuration ?

- **Propriétés** : *Connect Timeout, Read Timeout (ClientProperties)*
- **Filtres** : sur la requête *ClientRequestFilter* et sur la réponse *ClientResponseFilter*



# Développement client : 2 – cibler la ressource

- À partir d'une instance de *Client*, créer un objet *WebTarget* pour préciser l'URI de la ressource visée
- **Chemin racine**

```
WebTarget webTarget = client.target("http://127.0.0.1:9992/.../api/books");
```

- **Chemins intermédiaires (Template Parameters)**

```
WebTarget qpUri = webTarget.path("/books/queryparameters");
```

- **Paramètres de requêtes (Query Parameters)**

```
WebTarget book = qpURI.queryParam("name", "harry").queryParam("isbn", ...);
```

# Développement client : 3 – créer et invoquer la requête

- À partir d'une instance de *WebTarget*, créer une requête en précisant les types supportés par le client

```
Builder request = book.request(MediaType.APPLICATION_JSON_TYPE, MediaType.TEXT_PLAIN_TYPE)
```

- Enfin invoquer la requête en choisissant une méthode HTTP

- *<T> get(Class<T> c)* : GET avec un type de retour T
- *Response post(Entity<?> entity)* : POST avec un contenu
- *Response put(Entity<?> entity)* : PUT avec un contenu
- *Response delete(Entity<?> entity)* : DELETE avec un contenu

```
String value = request.get(String.class);
```

Peut retourner un type personnalisé ou *Response*



- Construire un objet *Entity* ?

- *Entity.entity(Object, MediaType)* : permet de transmettre un objet en précisant le type de média

```
Entity.entity(myBook, MediaType.APPLICATION_JSON_TYPE)
```

# Développement client

## ➤ Exemple : client pour récupérer un livre (GET)

```
public class LibraryRestWebServiceClientLauncher {  
  
    public LibraryRestWebServiceClientLauncher() throws IOException {  
        ResourceConfig resourceConfig = new ResourceConfig();  
        resourceConfig.registerClasses(BookResource.class, BookContentResource.class);  
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, resource);  
        server.start();  
  
        Client client = ClientBuilder.newClient();  
        String string = client.target(getBaseURI())  
            .path("books/details/123")  
            .request(MediaType.TEXT_PLAIN)  
            .get(String.class);  
        System.out.println(string);  
        string = client.target(getBaseURI())  
            .path("books/details/123")  
            .request(MediaType.TEXT_XML_TYPE)  
            .get(String.class);  
        System.out.println(string);  
        string = client.target(getBaseURI())  
            .path("books/details/123")  
            .request(MediaType.TEXT_HTML_TYPE)  
            .get(String.class);  
        System.out.println(string);  
        ...  
    }  
  
    private static URI getBaseURI() {  
        return UriBuilder.fromUri("http://localhost:8088/libraryrestwebservice/").build();  
    }  
}
```

```
@Path("/books")  
public class BookResource {  
    ...  
    @GET  
    @Path("details/{id}")  
    @Produces(MediaType.TEXT_PLAIN)  
    public String getDetailTextBookId(@PathParam("id") String id) {  
        return "Ce livre est une introduction sur la vie";  
    }  
    @GET  
    @Path("details/{id}")  
    @Produces(MediaType.TEXT_XML)  
    public String getDetailXMLBookId(@PathParam("id") String id) {  
        return "<?xml version=\"1.0\"?>" + ... + "</details>";  
    }  
    @GET  
    @Path("details/{id}")  
    @Produces(MediaType.TEXT_HTML)  
    public String getDetailHTMLBookId(@PathParam("id") String id) {  
        return "<html> " + ... + "</html> ";  
    }  
}
```

*BookResource.java*

*LibraryRestWebServiceClientLauncher.java* du  
projet **jaxrs-libraryrestwebservice**



# Développement client

- Exemple : client pour créer un livre (POST) depuis un contenu de type *String*

```
public class LibraryRestWebServiceClientLauncher {

    public LibraryRestWebServiceClientLauncher() throws IOException {
        ResourceConfig resourceConfig = new ResourceConfig();
        resourceConfig.registerClasses(BookResource.class, BookContentResource.class);
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, resourceConfig);
        server.start();

        Client client = ClientBuilder.newClient();
        String myBookString = "Le Livre";
        Response create = client.target(getBaseURI())
            .path("books")
            .request()
            .post(Entity.entity(myBookString, MediaType.TEXT_PLAIN_TYPE));
        System.out.println(create.getStatusInfo().getReasonPhrase());
        ...
    }

    private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost:8088/libraryrestwebbservice/").build();
    }
}
```

*LibraryRestWebServiceClientLauncher.java* du  
projet **jaxrs-libraryrestwebbservice**

```
@Path("/books")
public class BookResource {
    ...
    @POST
    public String createBook(String livre) {
        System.out.println("BookResource.createBook()");
        return livre;
    }
    ...
}
```

*BookResource.java*

# Développement client

- Exemple : client pour créer un livre (POST) depuis un contenu de type *String* via un formulaire

```
public class LibraryRestWebServiceClientLauncher {

    public LibraryRestWebServiceClientLauncher() throws IOException {
        ResourceConfig resourceConfig = new ResourceConfig();
        resourceConfig.registerClasses(BookResource.class, BookContentResource.class);
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, resourceConfig);
        server.start();

        Client client = ClientBuilder.newClient();
        String myBookString = "Le Livre";
        Form form = new Form();
        form.param("name", myBookString);

        Response create = client.target(getBaseURI())
            .path("books/createfromform")
            .request(MediaType.APPLICATION_FORM_URLENCODED)
            .post(Entity.form(form));
        System.out.println(create.getStatusInfo().getReasonPhrase());
        ...
    }
    ...
}
```

*LibraryRestWebServiceClientLauncher.java* du  
projet **jaxrs-libraryrestwebbservice**

```
@Path("/books")
public class BookResource {
    ...
    @POST
    @Path("createfromform")
    @Consumes("application/x-www-form-urlencoded")
    public String createBookFromForm(@FormParam("name") String name) {
        System.out.println("BookResource.createBookFromForm()");
        return name;
    }
    ...
}
```

*BookResource.java*

JAX-RS - M. Baron - Page 77



# Développement client

- Exemple : client pour récupérer un livre (GET) pour un contenu de type personnalisé

```
public class LibraryRestWebServiceClientLauncher {  
  
    public LibraryRestWebServiceClientLauncher() throws IOException {  
        ResourceConfig resourceConfig = new ResourceConfig();  
        resourceConfig.registerClasses(BookResource.class, BookContentResource.class);  
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, resourceConfig);  
        server.start();  
  
        Client client = ClientBuilder.newClient();  
        Book current = client.target(getBaseURI())  
            .path("contentbooks/json")  
            .request(MediaType.APPLICATION_JSON_TYPE)  
            .get(Book.class);  
  
        System.out.println(current);  
        ...  
    }  
  
    private static URI getBaseURI() {  
        return UriBuilder.fromUri("http://localhost:8088/libraryrestwebbservice/").build();  
    }  
}
```

En supposant qu'une classe  
*Book* a été créée côté client

```
@Path("/contentbooks")  
public class BookContentResource {  
    ...  
    @Path("json")  
    @GET  
    @Produces(MediaType.APPLICATION_JSON)  
    public Book getContentBookWithJSON() {  
        System.out.println("BookContentResource.getContentBookWithJSON()");  
        Book current = new Book();  
        current.setIsbn("1-111111-11");  
        current.setName("Harry");  
        return current;  
    }  
    ...  
}
```



# Développement client

- Exemple : client pour récupérer une collection de livres (GET) pour un contenu de type personnalisé

```
public class LibraryRestWebServiceClientLauncher {  
  
    public LibraryRestWebServiceClientLauncher() throws IOException {  
        ResourceConfig resourceConfig = new ResourceConfig();  
        resourceConfig.registerClasses(BookResource.class, BookContentResource.class);  
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, resourceConfig);  
        server.start();  
  
        Client client = ClientBuilder.newClient();  
        List<Book> books = client.target(getBaseURI())  
            .path("contentbooks")  
            .request(MediaType.APPLICATION_JSON_TYPE)  
            .get(new GenericType<List<Book>>() {});  
  
        System.out.println(books.size());  
        ...  
    }  
  
    private static URI getBaseURI() {  
        return UriBuilder.fromUri("http://localhost:8088/libraryrestwebbservice/").build();  
    }  
}
```

Utilisation de la classe  
*GenericType* pour préciser le  
type de retour

*LibraryRestWebServiceClientLauncher.java* du  
projet **jaxrs-libraryrestwebbservice**

```
@Path("/contentbooks")  
public class BookContentResource {  
    ...  
    @GET  
    @Produces(MediaType.APPLICATION_JSON)  
    public List<Book> getContentBooksWithJSON() {  
        System.out.println("BookContentResource.getContentBooksWithJSON()");  
        Book book1 = new Book();  
        current.setIsbn("1-111111-11");  
        current.setName("Harry");  
  
        Book book2 = ...  
        return Arrays.asList(book1, book2);  
    }  
}
```

*BookContentResource.java*



# Développement client

- Exemple : client pour mettre à jour un livre (PUT) depuis un contenu de type personnalisé

```
public class LibraryRestWebServiceClientLauncher {  
  
    public LibraryRestWebServiceClientLauncher() throws IOException {  
        ResourceConfig resourceConfig = new ResourceConfig();  
        resourceConfig.registerClasses(BookResource.class, BookContentResource.class);  
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, resourceConfig);  
        server.start();  
  
        Client client = ClientBuilder.newClient();  
        Book myBook = new Book();  
        myBook.setIsbn("1-111111-11");  
        myBook.setName("harry");  
        Response put = client.target(getBaseURI())  
            .path("contentbooks/json")  
            .request()  
            .put(Entity.entity(myBook, MediaType.APPLICATION_JSON_TYPE));  
  
        System.out.println(put.getStatusInfo().getReasonPhrase());  
        ...  
    }  
}
```

En supposant qu'une classe  
*Book* a été créée côté client

*LibraryRestWebServiceClientLauncher.java*  
du projet **jaxrs-libraryrestwebserice**

```
@Path("/contentbooks")  
public class BookContentResource {  
    ...  
    @Path("json")  
    @Consumes(MediaType.APPLICATION_JSON)  
    @PUT  
    public void updateContentBookWithJSON(Book current) throws IOException {  
        System.out.println("BookContentResource.updateContentBookWithJSON()");  
        System.out.println("Name: " + current.getName() +  
            ", ISBN: " + current.getIsbn());  
    }  
    ...  
}
```

*BookContentResource.java*

# Test d'intégration : tester ses services

- Tester les services JAX-RS consiste à s'assurer que le code du côté serveur est conforme
- Test d'intégration **VS** ➤ Test unitaire
  - 1) Démarrer un serveur
  - 2) Déployer les ressources
  - 3) Appeler les services web
  - 4) Vérifier les réponses (Assert)
  - 5) Arrêter le serveur
  - 1) Créer les bouchons (Mocks)
  - 2) Injecter les bouchons
  - 3) Invoquer les méthodes Java
  - 4) Vérifier le retour (Assert)
- Comment implémenter vos tests d'intégration ?
  - API cliente JAX-RS (comme vu précédemment)
    - Nécessite de faire manuellement les étapes 1 et 5
  - KarateDSL un framework basé sur Behaviour Driver Development pour écrire des tests sans avoir à utiliser Java
  - **Framework fourni par Jersey basé sur l'API cliente JAX-RS**
    - S'occupe des étapes 1 et 5

# Test d'intégration : tester ses services

## ➤ Ajout de dépendances Maven

Ne pas oublier le scope à *test* afin d'éviter de fournir ces dépendances lors de la mise en production

```
...
<dependency>
    <groupId>org.glassfish.jersey.test-framework</groupId>
    <artifactId>jersey-test-framework-core</artifactId>
    <scope>test</scope>
</dependency>

<dependency>
    <groupId>org.glassfish.jersey.test-framework.providers</groupId>
    <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
    <scope>test</scope>
</dependency>
...
```



Possibilité d'utiliser différents serveurs web (Jetty, Netty, InMemory, JDK...)

# Test d'intégration : tester ses services

## ➤ Exemple : vérifier les paramètres d'en-tête d'une réponse

```
public class BookResourceIntegrationTest extends JerseyTest {  
  
    @Override  
    protected Application configure() {  
        return new ResourceConfig(BookResource.class);  
    }  
  
    @Test  
    public void getHeaderParameterBookTest() {  
        // Given  
        String name = "harry";  
        String isbn = "1-111111-11";  
        boolean isExtended = true;  
  
        // When  
        Response response = target("/books/headerparameters")  
            .request()  
            .header("name", name)  
            .header("isbn", isbn)  
            .header("isExtended", isExtended).get();  
  
        // Then  
        Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),  
            response.getStatus());  
        String content = response.readEntity(String.class);  
        Assert.assertEquals("Content of response is: ", "harry 1-111111-11 true", content);  
    }  
}
```

The diagram illustrates the flow of the test logic. It starts with the `getHeaderParameterBookTest()` method in `BookResourceIntegrationTest.java`. A line points from the `target` call to the `BookResource` class definition. Another line points from the `get` call back to the `getHeaderParameterBook` method in `BookResource.java`.

*BookResourceIntegrationTest.java*  
du projet **jaxrs-libraryrestwebservice**

# Plan du cours

- Généralités JAX-RS
- Premier service web JAX-RS
- Développement serveur
  - Ressources : *@Path*
  - Méthodes : *@POST*, *@PUT*, *@DELETE* et *@GET*
  - Représentation : gestion du contenu et *Response*
- Développement client et test d'intégration
- **Déploiement**



# Déploiement

- Deux formes de déploiement pour exécuter votre service web
  - **Déploiement sur un serveur d'application Java**
    - Avant l'arrivée des microservices
    - Nécessite l'installation d'un serveur compatible ou pas Java EE (Jetty, WildFly, Glassfish...)
  - **Déploiement comme une application Java classique**
    - Populaire depuis l'arrivée des microservices
    - Serveur d'application est intégré (embedded)

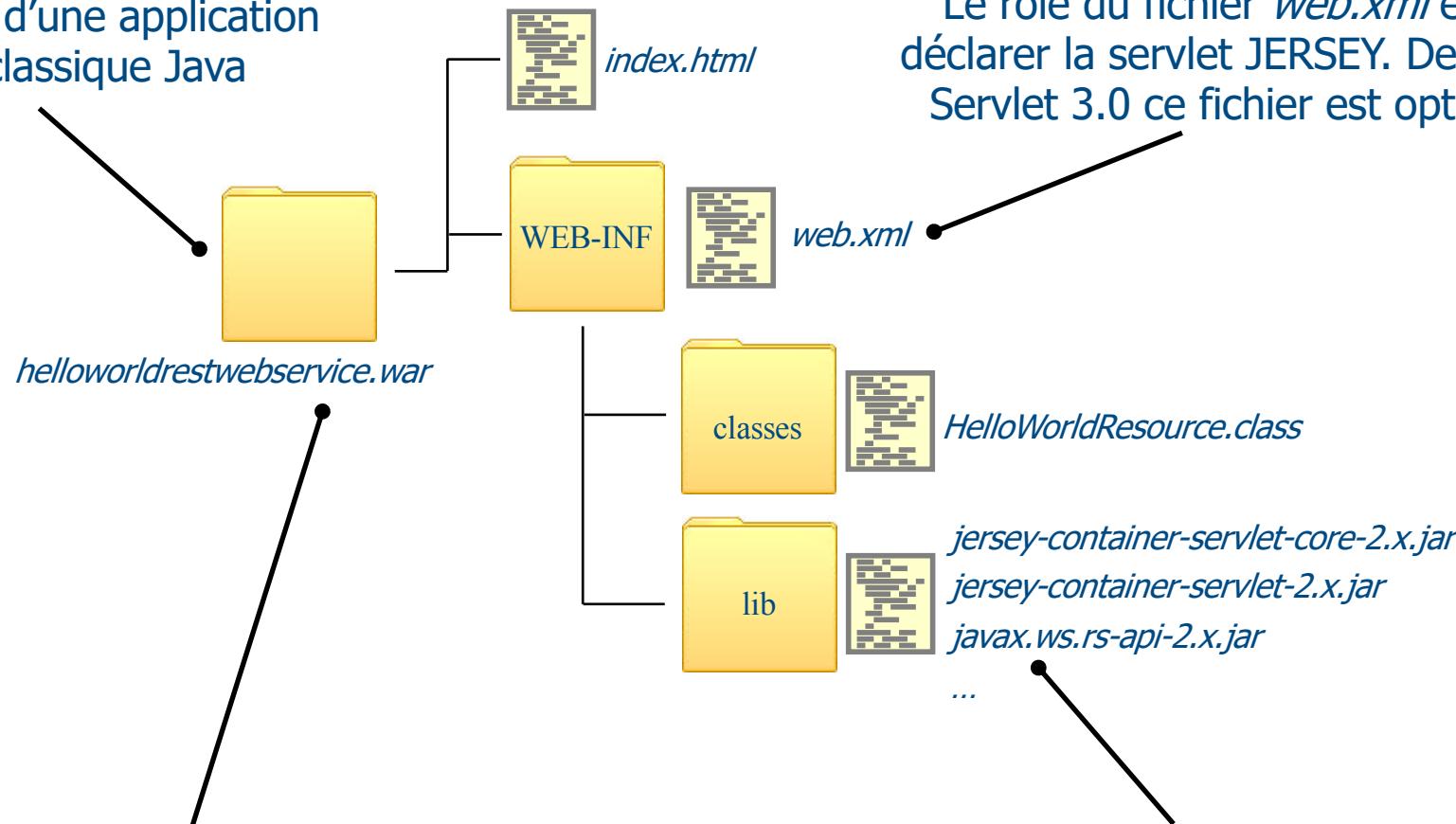
# Déploiement : configuration

- L'objectif de la configuration est d'identifier les **ressources** qui seront **actives** lors du déploiement
- La classe *Application* permet de préciser les *ressources*
  - *Set<Class> getClasses()* : classes des ressources
  - *Set<Object> getSingletons()* : instances des ressources
- *Application* est une implémentation à vide, la classe *ResourceConfig* fournit une implémentation plus complète
- Dans les deux cas de déploiement, il faudra fournir une instance de *Application*

# Déploiement : serveur d'application Java

## ➤ Structure du service web REST « HelloWorldRestWebService »

Structure d'une application web classique Java



Le rôle du fichier `web.xml` est de déclarer la servlet JERSEY. Depuis les Servlet 3.0 ce fichier est optionnel

Cette application web contient un service web REST

Les bibliothèques ne sont pas obligatoires pour les serveurs d'application compatibles Java EE (ex. Glassfish)



## Déploiement : serveur d'application Java

- Les applications JAX-RS sont construites et déployées sous le format d'une application web Java (war)
- La configuration de JAX-RS déclare les classes ressources par l'intermédiaire du fichier de déploiement (web.xml)
- Trois types de configuration sont autorisées
  1. Contenu *web.xml* classique
  2. Contenu *web.xml* pointe vers *Application/ResourceConfig*
  3. Seulement avec *Application/ResourceConfig*

# Déploiement : serveur d'application Java

## ➤ Exemple 1 : déclaration des ressources depuis web.xml

Servlet fournie par Jersey pour le traitement des requêtes HTTP

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" ...>
    <display-name>HelloWorldRestWebService</display-name>
    <servlet>
        <servlet-name>jersey-servlet</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.providerclassnames</param-name>
            <param-value>
                fr.mickaelbaron.helloworldrestwebservice.HelloWorldResource
            </param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
</web-app>
```

*onlywebclasses.xml/ du projet*

**jaxrs-helloworldrestwebservicefromwar**

# Déploiement : serveur d'application Java

- Exemple 1 bis : déclaration des packages contenant les ressources depuis web.xml

Servlet fournie par Jersey pour le traitement des requêtes HTTP

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" ...>
    <display-name>HelloWorldRestWebService</display-name>
    <servlet>
        <servlet-name>jersey-servlet</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>jersey.config.server.provider.packages</param-name>
            <param-value>
                fr.mickaelbaron.helloworldrestwebservice
            </param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
</web-app>
```

*onlywebpackages.xml du projet*

**jaxrs-helloworldrestwebservicefromwar**

# Déploiement : serveur d'application Java

- Exemple 2 : déclaration de ressources avec *ResourceConfig* depuis web.xml

Servlet fournie par Jersey pour le traitement des requêtes HTTP

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="3.1" ...>
    <display-name>HelloWorldRestWebService</display-name>
    <servlet>
        <servlet-name>jersey-servlet</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
            <param-name>javax.ws.rs.Application</param-name>
            <param-value>
                fr.mickaelbaron.helloworldrestwebservice.HelloWorldApplication
            </param-value>
        </init-param>
    </servlet>
    <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/api/*</url-pattern>
    </servlet-mapping>
</web-app>
```

webapplication.xml

```
public class HelloWorldApplication extends ResourceConfig {
    public HelloWorldApplication() {
        this.packages("fr.mickaelbaron.helloworldrestwebservice");
    }
}
```

HelloWorldApplication.java du projet  
jaxrs-helloworldrestwebservicefromwar

Déclaration du package qui contiendra les ressources à remonter



# Déploiement : serveur d'application Java

- Exemple 3 : déclaration de ressources avec *ResourceConfig* sans web.xml

```
@ApplicationPath("api")
public class HelloWorldApplication extends ResourceConfig {

    public HelloWorldApplication() {
        this.packages("fr.mickaelbaron.helloworldrestwebservice");
    }
}
```

*HelloWorldApplication.java* du projet  
**jaxrs-helloworldrestwebservicefromwar**

Le serveur d'application doit être compatible avec Servlet 3

# Déploiement : serveur d'application Java

- Exemple : déployer un service web (packagé war) dans une instance Tomcat via Docker

```
$ mvn clean package -P war-withoutweb # Compile et build le projet jaxrs-helloworldrestwebservicefromwar en  
invoquant le profil war-withoutweb (exemple 3 précédent)  
=> Fichier helloworldwebservice.war disponible dans le répertoire target/  
  
$ docker pull tomcat:9-jre11-slim # Télécharge la version 9 de Tomcat avec une JRE 11  
=> Image Docker disponible  
  
$ docker run --rm --name helloworldrestservice-tomcat -v $(pwd)/target/helloworldrestwebservicefromwar.war:/usr/  
    local/tomcat/webapps/helloworldrestwebservicefromwar.war -it -p 8080:8080 tomcat:9-jre11-slim  
=> Service web disponible à l'adresse http://localhost:8080/helloworldrestwebservice/api/hello
```



# Déploiement : application Java classique

- JAX-RS peut être déployée comme une application Java (JAR) sans avoir à fournir une application web (WAR)
- À la différence de JAX-WS l'implémentation Jersey nécessite l'ajout d'un serveur web en mode embarqué
  - **Grizzly (le serveur web de Glassfish)**
  - Undertow (le serveur web de Wildfly)
  - Jetty
  - Tomcat...
- Usages
  - Pour les tests fonctionnels, fournir des *bouchons* de services web
  - Déployer son application comme un microservice (voir cours)
- Le développement des services web reste identique
- L'appel des services web (client) ne nécessite pas de configuration particulière

# Déploiement : application Java classique

- Le serveur web Grizzly supporte l'API NIO permettant de traiter de nombreuses requêtes en parallèle
- Grizzly est le serveur web derrière Glassfish
- Grizzly est utilisé par Jersey pour ses tests fonctionnels
- Dépendances Maven à ajouter

```
<dependency>
    <groupId>org.glassfish.jersey.containers</groupId>
    <artifactId>jersey-container-grizzly2-http</artifactId>
</dependency>
```

```
<dependency>
    <groupId>com.sun.xml.bind</groupId>
    <artifactId>jaxb-impl</artifactId>
    <version>2.3.1</version>
</dependency>
<dependency>
    <groupId>com.sun.istack</groupId>
    <artifactId>istack-commons-runtime</artifactId>
    <version>3.0.7</version>
</dependency>
<dependency>
    <groupId>com.sun.xml.txw2</groupId>
    <artifactId>txw2</artifactId>
    <version>20110809</version>
</dependency>
```

Nécessaire quand des types personnalisés sont sérialisés



Utilisées pour la génération du document WADL

# Déploiement : application Java classique

## ➤ Exemple : utiliser JAX-RS avec Grizzly

```
public class HelloWorldRestWebServiceLauncher {  
  
    public static final URI BASE_URI = getBaseURI();  
  
    private static URI getBaseURI() {  
        return UriBuilder.fromUri("http://localhost/helloworldrestbservice/api/").port(9992).build();  
    }  
  
    @Test  
    public static void main(String[] args) throws ... {  
        ResourceConfig resourceConfig = new ResourceConfig();  
        resourceConfig.registerClasses(HelloWorldResource.class);  
  
        HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, resourceConfig);  
        server.start();  
  
        System.out.println(String.format("Jersey app started with WADL available at "  
            + "%sapplication.wadl\nHit enter to stop it...",  
            BASE_URI, BASE_URI));  
  
        System.in.read();  
        server.shutdownNow();  
    }  
}
```

Configuration pour accéder aux classes ressources

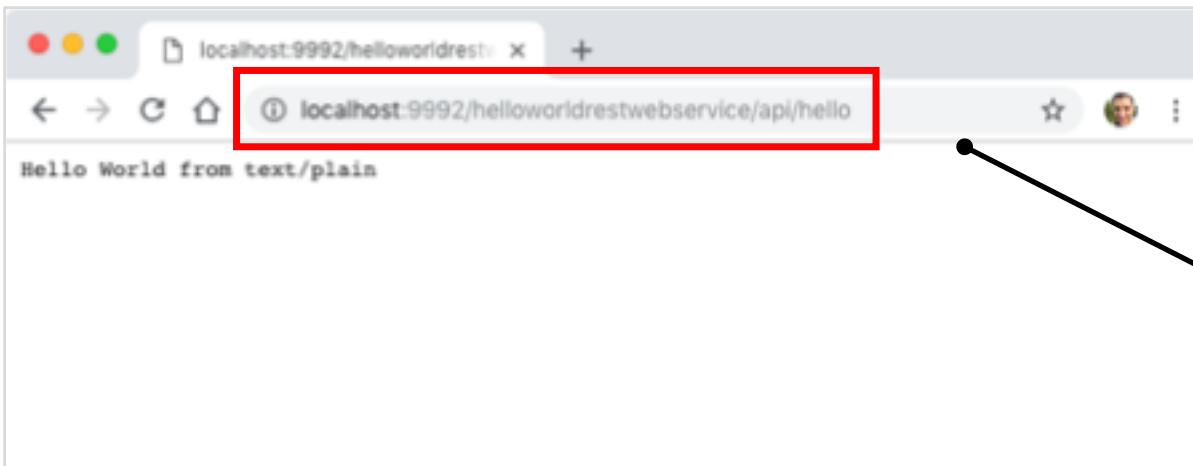
Création d'une instance du serveur Web

*HelloWorldRestWebServiceLauncher.java du projet  
jaxrs-helloworldrestbservice*



# Déploiement : application Java classique

- Exemple (suite) : utiliser JAX-RS avec Grizzly



Résultat identique qu'une application déployée dans un serveur

```
<application xmlns="http://wadl.dev.java.net/2009/02">
  <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.27 2018-04-15T07:34:57Z"/>
  <doc xmlns:jersey="http://jersey.java.net/" jersey:hint="This is simplified WADL with user and core resources only. To get full WADL with extended resources use the query parameter detail. Link: http://localhost:9992/helloworldrestwebservice/api/application.wadl?detail=true"/>
  <grammars/>
  <resources base="http://localhost:9992/helloworldrestwebservice/api/">
    <resource path="/hello">
      <method id="getHelloWorld" name="GET">
        <response>
          <representation mediaType="text/plain"/>
        </response>
      </method>
    </resource>
  </resources>
</application>
```

A screenshot of a web browser window. The address bar shows the URL "localhost:9992/helloworldrestwebservice/api/application.wadl". This URL is highlighted with a red rectangular box. Below the address bar, the page content displays the XML code of the WADL document. A black arrow points from the text "Document WADL généré automatiquement /application.wadl" to the red box.

Document WADL généré automatiquement  
*/application.wadl*