

## Services Web RESTFull (1/4)

### a. Questions :

Avant de commencer, essayez de répondre à ces questions :

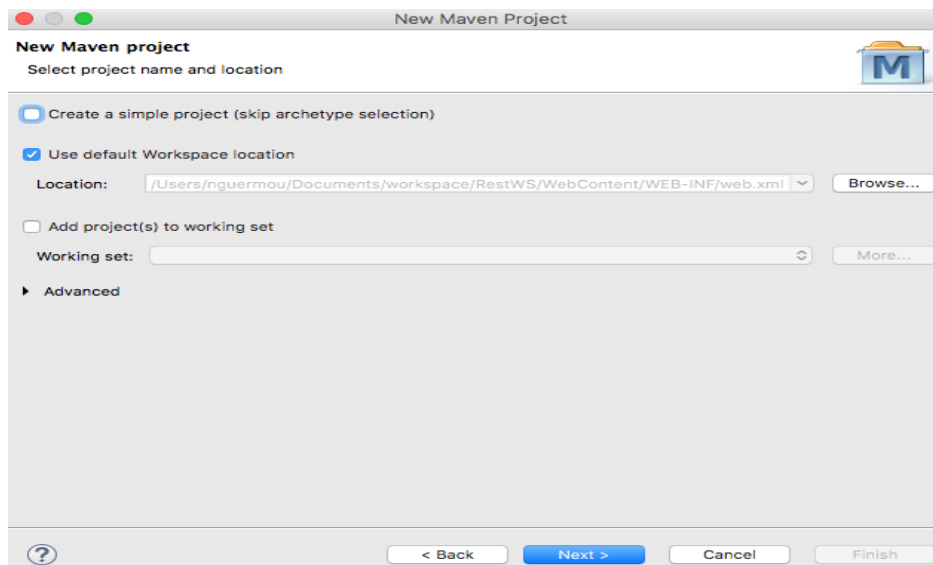
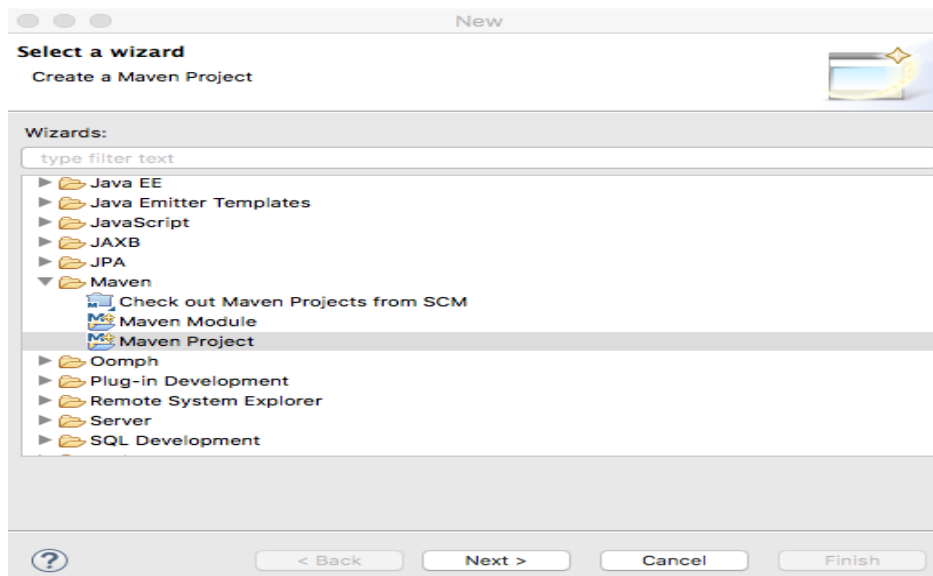
- C'est quoi une architecture orientée ressource ?
- C'est quoi un service RestFull?
- C'est quoi la différence entre les services Web SOAP et Rest ?

### b. Compétences visées :

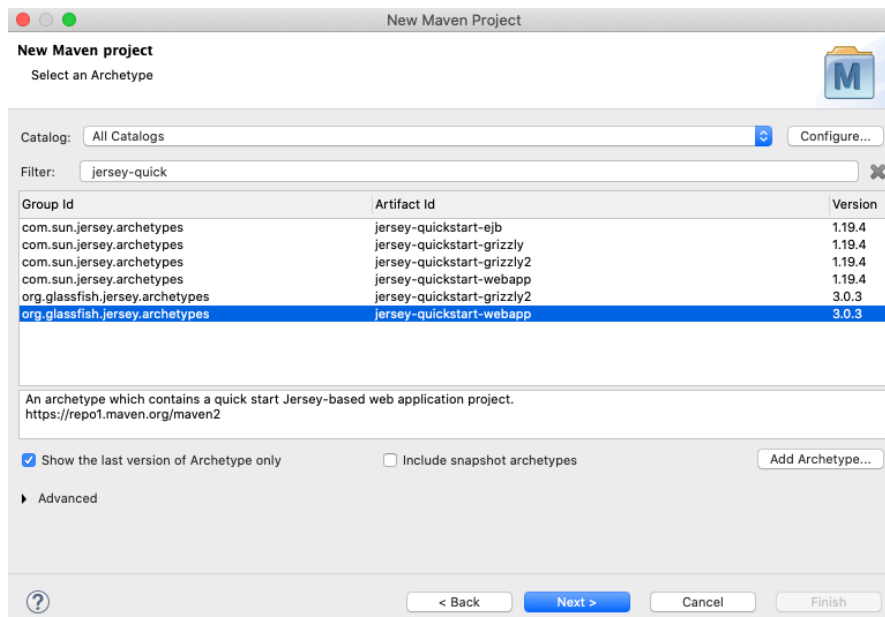
En suivant les étapes suivantes, vous devriez être capable de : (1) créer un service Web Rest à partir d'une classe Java en utilisant Maven et (2) de le tester.

### c. Création d'un projet Maven :

Créez un projet Maven

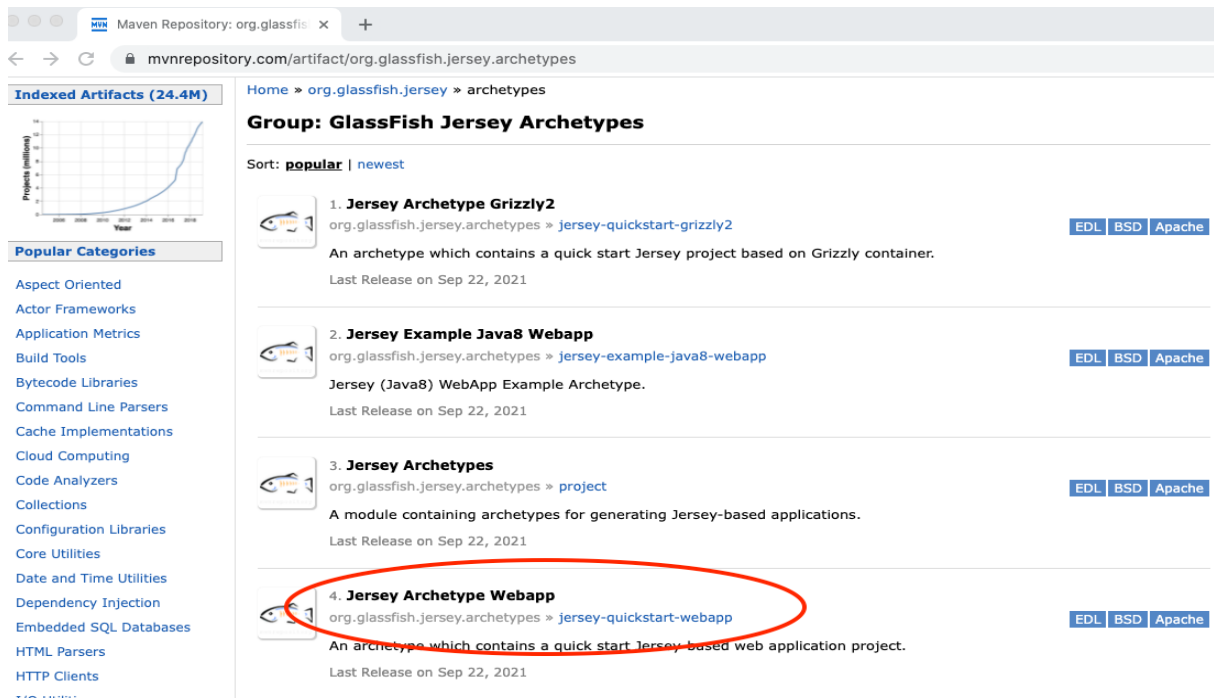


Cherchez dans la liste ou en utilisant le Filter « *Jersey-quick* » et sélectionnez-le (jersey-quickstart-webapp de org.glassfish.jersey). Si vous ne le trouvez pas, vous allez pouvoir l'ajouter. Pour cela passez à la section **Ajout de Jersey**.



### Ajout de Jersey s'il est absent :

Pour ajouter Jersey, vous allez récupérer la dépendance Jersey depuis le site de Maven. Le moyen le plus rapide c'est de chercher « Maven Jersey Archetype » sur google.



Home » [org.glassfish.jersey.archetypes](#) » [jersey-quickstart-webapp](#)

### Jersey Archetype Webapp

An archetype which contains a quick start Jersey-based web application project.

**License** [Apache 2.0](#) [BSD 2-clause](#) [EDL 1.0](#) [EPL 2.0](#) [MIT](#) [Public](#) [W3C](#)

**Categories** [Maven Archetypes](#)

**Tags** [archetype](#) [webapp](#) [maven](#) [webservice](#) [example](#)

Central (97)

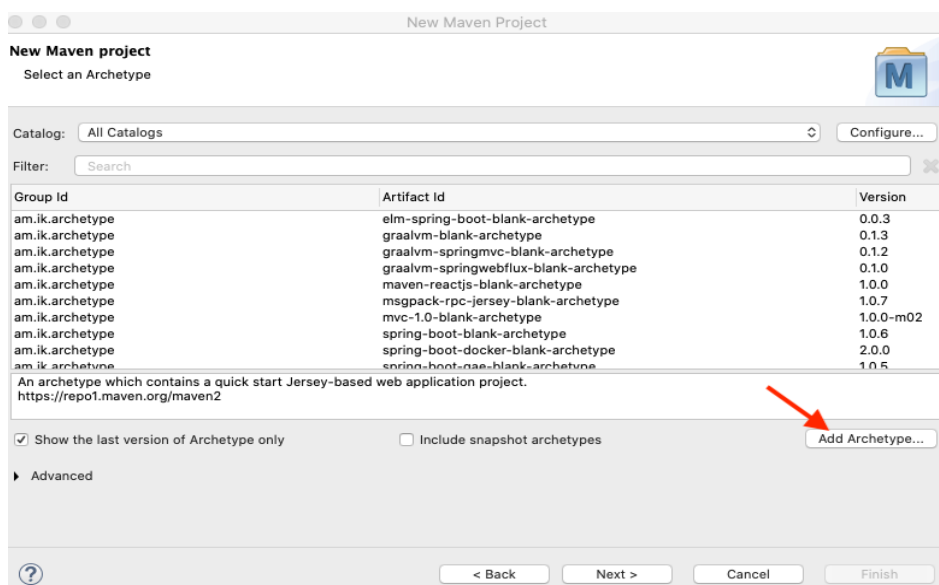
	Version	Repository	Usages	Date
3.0.x	<a href="#">3.0.3</a>	Central	0	Sep, 2021
	<a href="#">3.0.2</a>	Central	0	Apr, 2021
	<a href="#">3.0.1</a>	Central	0	Jan, 2021
	<a href="#">3.0.0</a>	Central	0	Dec, 2020
	<a href="#">3.0.0-RC2</a>	Central	0	Nov, 2020
	<a href="#">3.0.0-M6</a>	Central	0	Jun, 2020
	<a href="#">3.0.0-M1</a>	Central	0	Apr, 2020
2.35.x	<a href="#">2.35</a>	Central	0	Sep, 2021
2.34.x	<a href="#">2.34</a>	Central	0	Apr, 2021
2.33.x	<a href="#">2.33</a>	Central	0	Dec, 2020
2.32.x	<a href="#">2.32</a>	Central	0	Sep, 2020
2.31.x	<a href="#">2.31</a>	Central	0	May, 2020

Vous pouvez choisir la version 3.0.3. En cliquant dessus, on obtient les informations nécessaires pour ajouter la dépendance Maven à votre Eclipse :

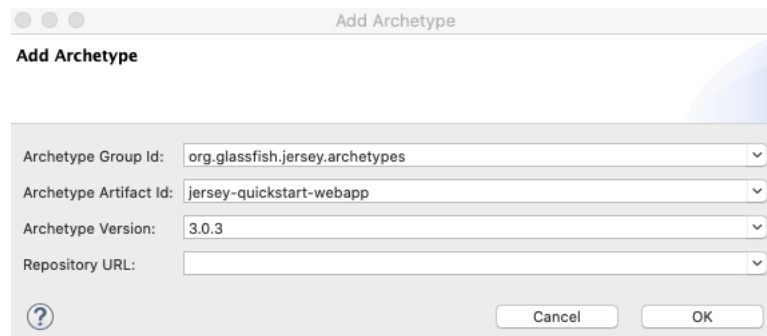
Maven [Gradle](#) [Gradle \(Short\)](#) [Gradle \(Kotlin\)](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
<!-- https://mvnrepository.com/artifact/org.glassfish.jersey.archetypes/jersey-quickstart-webapp -->
<dependency>
  <groupId>org.glassfish.jersey.archetypes</groupId>
  <artifactId>jersey-quickstart-webapp</artifactId>
  <version>3.0.3</version>
</dependency>
```

Vous allez ajouter la dépendance à votre Eclipse. Pour cela, reprenez votre Eclipse et cliquez sur « Add Archetype ».

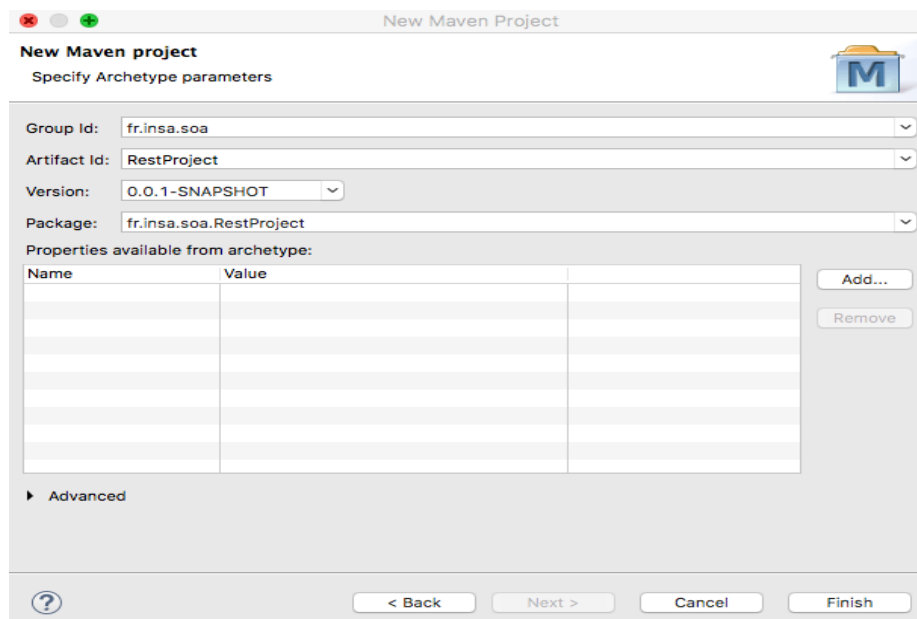


Remplissez les différents champs pour créer la dépendance récupérée précédemment :



### Suite de la création du projet :

Créez votre projet jersey webapp. Renseignez les informations de votre projet :



Le projet *RestProject* est créé. Il contient par défaut un service Rest que vous allez examiner ultérieurement.

Si le fichier pom.xml de votre projet contient l'erreur suivante :



Ouvrez le fichier pom.xml de votre projet et ajoutez le code suivant sous l'élément plugins. Cela va permettre d'ajouter le plugin *Maven War*, responsable de récupérer

toutes les dépendances, les classes et ressources nécessaires pour votre projet lors de la création du paquet de déploiement (i.e., le paquet war) de votre projet.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-war-plugin</artifactId>
  <version>3.3.1</version>
</plugin>
```

Par défaut, c'est la version 1.8 de Java qui est utilisée :

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.insa.soa</groupId>
  <artifactId>RestProject</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>RestProject</name>
  <build>
    <finalName>RestProject</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <inherited>true</inherited>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <version>3.3.1</version>
      </plugin>
    </plugins>
  </build>
```

Si vous le souhaitez, vous pouvez mettre à jour votre JDK à 11 par exemple :

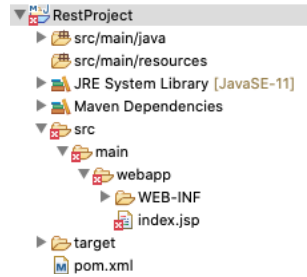
```
<build>
  <finalName>RestProject</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.8.1</version>
      <inherited>true</inherited>
      <configuration>
        <source>11</source>
        <target>11</target>
      </configuration>
    </plugin>
  </plugins>
```

Aussi, mettez à jour la version de jersey (utilisez la version 3.0.0-M1) sous l'élément *properties* :

```
<properties>
  <jersey.version>3.0.0-M1</jersey.version>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
```

Mettez à jour votre projet (clic droit sur le projet, puis *Maven*, puis *Update Project*). Cela va permettre de considérer les modifications apportées au fichier pom.xml.

Si des erreurs au niveau de la page index.jsp sont signalées :



Ouvrez la page *index.jsp*. La classe `HttpServlet` (que le serveur tomcat utilise pour intercepter des requêtes http) est manquante.

```
1 The superclass "jakarta.servlet.http.HttpServlet" was not found on the Java Build Path
2 <body>
3   <h2>Jersey RESTful Web Application!</h2>
4   <p><a href="webapi/myresource">Jersey resource</a>
5   <p>Visit <a href="http://jersey.java.net">Project Jersey website</a>
6   for more information on Jersey!
7 </body>
```

Ajoutez la dépendance correspondante :

```
<dependency>
  <groupId>jakarta.servlet</groupId>
  <artifactId>jakarta.servlet-api</artifactId>
  <version>5.0.0</version>
  <scope>provided</scope>
</dependency>
```

Comme vous allez utiliser Json pour la sérialisation des données ultérieurement, décommentez la dépendance de Json :

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

Votre fichier pom.xml est comme suit maintenant :

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-
v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>fr.insa.soa</groupId>
  <artifactId>RestProject</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>
  <name>RestProject</name>

  <build>
    <finalName>RestProject</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <inherited>true</inherited>
        <configuration>
```

```
        <source>11</source>
        <target>11</target>
    </configuration>
</plugin>
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-war-plugin</artifactId>
    <version>3.3.1</version>
</plugin>
</plugins>
</build>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>org.glassfish.jersey</groupId>
            <artifactId>jersey-bom</artifactId>
            <version>${jersey.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>

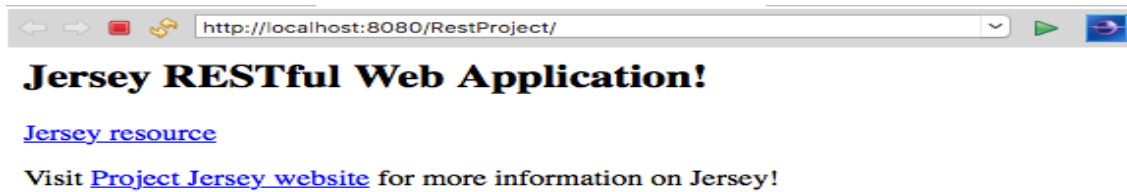
<dependencies>
    <dependency>
        <groupId>org.glassfish.jersey.containers</groupId>
        <artifactId>jersey-container-servlet-core</artifactId>
        <!-- use the following artifactId if you don't need servlet 2.x compatibility -->
        <!-- artifactId>jersey-container-servlet</artifactId -->
    </dependency>

    <dependency>
        <groupId>org.glassfish.jersey.inject</groupId>
        <artifactId>jersey-hk2</artifactId>
    </dependency>

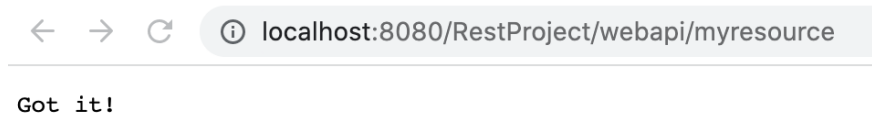
    <dependency>
        <groupId>org.glassfish.jersey.media</groupId>
        <artifactId>jersey-media-json-binding</artifactId>
    </dependency>
    <dependency>
        <groupId>jakarta.servlet</groupId>
        <artifactId>jakarta.servlet-api</artifactId>
        <version>5.0.0</version>
        <scope>provided</scope>
    </dependency>
</dependencies>
<properties>
    <jersey.version>3.0.0-M1</jersey.version>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
</project>
```

Mettez à jour votre projet Maven pour que toutes les modifications soient prises en compte.

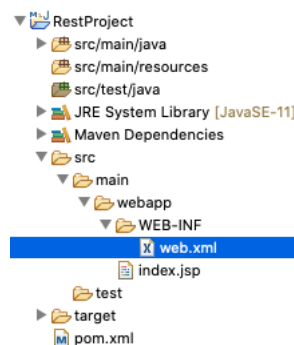
Déployez le projet sur le serveur Tomcat pour le tester (clic droit sur le projet/Run As/Run on Server). Ouvrez via un navigateur l'URL <http://localhost:8080/RestProject>



Cliquez sur *Jersey resource*. Une requête GET va être envoyée :



En regardant l'URL, à priori, le service (la ressource), s'appelle *myresource* et le contexte est *webapi* qui est défini dans le fichier *web.xml* sous *src/main/webapp/WEB-INF*.



Essayez de comprendre le lien entre l'URL et votre service.

### Remarque :

Si vous avez l'erreur suivante dans votre fichier *web.xml* (qui ne gêne pas l'exécution de votre projet) :



Ajoutez « ; » entre les deux URI de *xsi:schemaLocation* comme suit

```
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee; http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">
```

Le fichier *web.xml* est comme suit :

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This web.xml file is not required when using Servlet 3.0 container,
see implementation details http://jersey.java.net/nonav/documentation/latest/jax-rs.html
-->
```



```
<web-app version="2.5" xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee; http://java.sun.com/xml/ns/javaee/web-
app_2_5.xsd">
  <servlet>
    <servlet-name>Jersey Web Application</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
    <init-param>
      <param-name>jersey.config.server.provider.packages</param-name>
      <param-value>fr.insa.soa.RestProject</param-value>
    </init-param>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>Jersey Web Application</servlet-name>
    <url-pattern>/webapi/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Vous pouvez examiner le service dans le projet RestProject/src/main/java. Essayez de l'expliquer.

#### d. Création d'un service Web RESTful

Créez une classe, appelée ici *Comparator*. Dans le code source donné ci-après, vous remarquerez l'annotation `@Path("comparator")`. Le service **sera exposé sous forme d'une ressource *comparator***. Le service dispose d'une méthode `sayHello()`. Cette méthode sera appelée avec la méthode **http GET** (annotation `@GET`). Finalement, on indique que le résultat de cette méthode sera envoyé sous forme de texte (notation `@Produces(MediaType.TEXT_PLAIN)`).

```
package fr.insa.soa.RestProject;

import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

@Path("comparator")
public class Comparator {
    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello() {
        return "Bonjour!";
    }
}
```

Déployez votre projet, et testez-le via une requête http de type GET. Comment vous devriez le tester ?



Maintenant, vous allez ajouter une méthode qui reçoit une chaîne de caractère et qui renvoie sa taille (ici on l'appelle `getLongueur`). Rappelez-vous, tout est vu comme ressource. La méthode `getLongueur(String chaîne)` sera vue comme une ressource *longueur*. Le paramètre d'entrée chaîne est vu comme une ressource *chaîne*.

```
package fr.insa.soa.rest;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

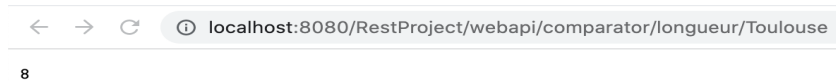
@Path("comparator")
public class Comparator {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String getLongueur(){
        return "Bonjour!";
    }

    @GET
    @Path("longueur/{chaine}")
    @Produces(MediaType.TEXT_PLAIN)
    public int getLongueur(@PathParam("chaine") String chaine){
        return chaine.length();
    }
}
```

A votre avis, pour utiliser la méthode `getLongueur(chaine)`, quelle URL vous devriez utiliser ? Prenez quelques minutes pour y réfléchir. N'hésitez pas à tenter !

Voici l'URL qu'il faut utiliser pour appeler la méthode `getLongueur(chaine)`. En effet, cette méthode correspond à la ressource *longueur* via laquelle on peut attribuer une valeur à la ressource *chaine* (.../longueur/**Toulouse**)



localhost:8080/RestProject/webapi/comparator/longueur/Toulouse

8

Vous allez ajouter une autre méthode et utiliser une autre annotation (`@QueryParam`), qui vous permet de faire passer des paramètres d'entrée d'une autre manière :

```
@GET
@Path("longueurDouble")
@Produces(MediaType.TEXT_PLAIN)
public int getLongueurDouble(@QueryParam("chaine") String chaine) {
    return chaine.length()*2;
}
```

Essayez l'URL en suivant le même principe que précédemment.

localhost:8080/RestProject/webapi/comparator/longueurDouble/Toulouse

Qu'est-ce que vous obtenez comme résultat ? Avez-vous une explication au résultat obtenu ?

La ressource n'est pas trouvée. C'est normal. Avec l'annotation *QueryParam*, le passage des paramètres se fait comme suivant :



localhost:8080/RestProject/webapi/comparator/longueurDouble?chaine=Toulouse

16

Vous allez ajouter une méthode à votre classe permettant de mettre à jour l'attribut *chaine*. Quelle opération (i.e., quel verbe http ?) il faut utiliser ?

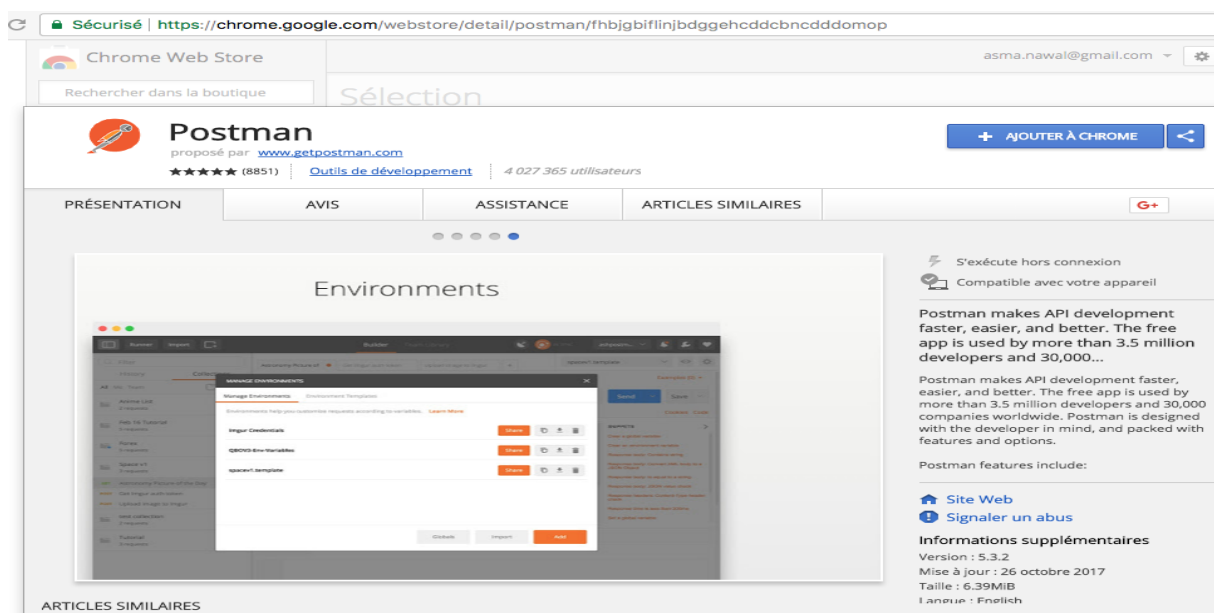
C'est bien la méthode PUT qu'il faut utiliser. Si vous n'étiez pas capable de répondre à cette question, il faut revoir le cours sur l'architecture Rest.

```
@PUT
@Path("/{idEtudiant}")
@Consumes(MediaType.TEXT_PLAIN)
public void updateEtudiant(@PathParam("idEtudiant") int id){
    System.out.println("mise à jour réussie!!!");
}
```

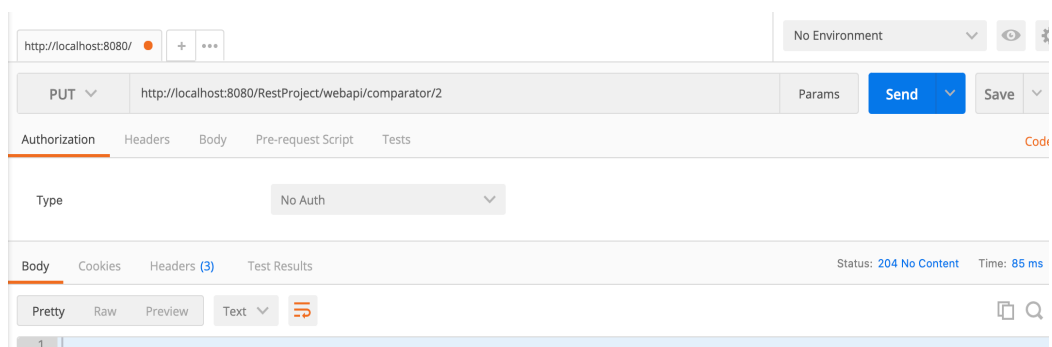
Pour tester votre méthode PUT, vous allez utiliser un client. En effet, avec un navigateur, **vous ne pouvez envoyer que des requêtes http de type GET**. Dans la section suivante, vous allez utiliser Postman.

### e. Utilisation du client Rest Postman :

Pour tester des requêtes Rest, notamment Post, Put, et Delete, vous avez besoin d'utiliser un client Rest. Ici, vous allez utiliser Postman de chrome. Cherchez postman chrome sur google.



Pour tester votre méthode PUT, utilisez Postman.



Vous pourrez vérifier que la requête PUT s'est bien exécutée en vérifiant votre console (message affiché).

Modifiez votre méthode PUT pour qu'elle renvoie un entier :

```
@PUT
@Path("/{idEtudiant}")
@Produces(MediaType.TEXT_PLAIN)
public int updateEtudiant(@PathParam("idEtudiant") int id) {
    System.out.println("Mise à jour réussie!");
    return id;
}
```

Tester votre méthode. Vous devriez avoir l'id envoyé.