Tutoriel sur le développement de services web REST avec JAX-RS, Maven et Eclipse

Par Mickael Baron

Date de publication : 25 janvier 2016

Dernière mise à jour : 2 février 2019

L'objectif de cette troisième leçon est d'apprendre à manipuler l'API JAX-RS pour le développement de services web REST à partir de la plateforme de développement Java.

Chaque exercice est fourni dans un dossier avec à l'intérieur un projet Java Maven contenant des classes et des fichiers de configuration qu'il faudra compléter au fur et à mesure des questions.

Buts pédagogiques : transformation d'une classe Java en service web REST, manipulation des annotations JAX-RS, tests d'intégration, client Java, implémentation Jersey, invocation des services REST via **cURL**, compilation avec Maven, déploiement du service web REST avec un serveur d'applications lui-même déployé dans un conteneur Docker, déploiement du service web REST depuis une classe principale Java.

Les codes pour les exercices sont disponibles sur le dépôt Git suivant : **https://github.com/mickaelbaron/jaxrs-tutorial** (pour récupérer le code faire : `$ git clone https://github.com/mickaelbaron/jaxrs-tutorial`).

La solution de tous les exercices est disponible sur la branch *solution :* `$ git checkout solution`.

Pour réagir au contenu de ce tutoriel, un espace de dialogue vous est proposé sur le forum : **Commentez**.

Prérequis logiciels....3
I - Exercice 1 : développer un service web REST « Bonjour ENSMA »....3
I-A - But....3
I-B - Description....3
I-C - Étapes à suivre....3
II - Exercice 2 : développer un service web REST « Interrogation et réservation de billets de train »....6
II-A - But....6
II-B - Description....6
II-C - Étapes à suivre....7
III - Exercice 3 : tests d'intégration de service web REST « Interrogation et réservation de billets de train »....16
III-A - But....16
III-B - Description....16
III-C - Étapes à suivre....16
IV - Exercice 4 : client de service web REST « Interrogation et réservation de billets de train »....22
IV-A - But....22
IV-B - Description....22
IV-C - Étapes à suivre....22
V - Exercice 5 : déployer le service web REST « Interrogation et réservation de billets de train »....25
V-A - But....25
V-B - Description....25
V-C - Étapes à suivre pour effectuer un déploiement comme une application Java classique....25
V-D - Étapes à suivre pour effectuer un déploiement sur le serveur d'applications Tomcat....26
VI - Conclusion et remerciements....28

Prérequis logiciels

Avant de démarrer cette série d'exercices sur l'utilisation de l'API JAX-WS, veuillez préparer votre environnement de développement en installant les outils suivants :

- **Java 8 à 11** ;
- **Maven** ;
- **Eclipse** ;
- **cURL** ;
- **Docker (exercice 5)**.

I - Exercice 1 : développer un service web REST « Bonjour ENSMA »

I-A - But

- Développer un service web REST à partir d'une classe Java.
- Déployer le service web REST comme une application Java classique.
- Afficher le contrat de description WADL.
- Tester le service web REST avec **cURL**.

I-B - Description

Le service web REST de ce premier exercice fournit un accès à la ressource « Hello » qui permet de retourner des messages de bienvenue. Le service proposé n'autorise que la lecture à la ressource (GET) et des paramètres peuvent être autorisés. Le format supporté par la ressource « Hello » sera du texte (text/plain).

I-C - Étapes à suivre

- Démarrer l'environnement de développement Eclipse.
- Importer le projet Maven **jaxrs-tutorial-exercice1** (**File -> Import -> Maven -> Existing Maven Projects**), choisir le répertoire du projet, puis faire **Finish**.
- Créer une classe qui représentera la ressource « Hello » (**File -> New** puis choisir **Class**). Appeler la classe HelloResource et la définir dans le package fr.mickaelbaron.jaxrstutorialexercice1.
- Dans la nouvelle classe créée, ajouter l'annotation @Path("hello") pour préciser que la ressource sera accessible via le chemin */hello* et l'annotation @Produces(MediaType.TEXT_PLAIN) pour indiquer que le contenu retourné au client sera de type texte (text/plain).

```
1. @Path("hello")
2. @Produces(MediaType.TEXT_PLAIN)
3. public class HelloResource {
4.     public HelloResource() { }
5. }
```

- Ajouter une première méthode String getHello() permettant de retourner une constante de type chaîne de caractères Bonjour ENSMA (ou autre texte de votre création).

```
1.     @GET
2.     public String getHello() {
3.         return "Bonjour ENSMA";
4.     }
```

- Afin de résoudre les problèmes de dépendances vers la bibliothèque JAX-RS, compléter le fichier de description Maven *pom.xml* en ajoutant la dépendance suivante (balise <dependencies>).

```
<dependency>
    <groupId>javax.ws.rs</groupId>
    <artifactId>javax.ws.rs-api</artifactId>
    <version>${jaxrs.version}</version>
</dependency>
```

Cette dépendance (API de JAX-RS) ne sert qu'à résoudre les problèmes de visibilité des annotations. Nous aurons besoin de dépendances supplémentaires (implémentation de JAX-RS) quand nous souhaiterons déployer notre service web REST.

- Compléter la classe HelloLauncher dans le package fr.mickaelbaron.jaxrstutorialexercice1. Cette classe sera utilisée pour publier localement notre service web REST.

```
public class HelloLauncher {

    public static final URI BASE_URI = getBaseURI();

    private static URI getBaseURI() {
        return UriBuilder.fromUri("http://localhost/api/").port(9991).build();
    }

    public static void main(String[] args) {
         ResourceConfig rc = new ResourceConfig();
         rc.registerClasses(HelloResource.class);
         rc.property(LoggingFeature.LOGGING_FEATURE_LOGGER_LEVEL_SERVER,
 Level.WARNING.getName());

         try {
             HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, rc);
             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();
         } catch (Exception e) {
             e.printStackTrace();
         }
     }
}
```

- Afin de résoudre les problèmes de dépendances vers le serveur Grizzly, compléter le fichier de description Maven *pom.xml*.

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

- Exécuter la classe HelloLauncher.
- Ouvrir une fenêtre d'un navigateur web et tester la récupération de la ressource « Hello » (requête GET via l'URL **http://localhost:9991/api/hello**).

Ce service web REST n'est pas complet puisqu'il n'est pas possible de paramétrer le message de bienvenue (utilisation de *template parameter*) ni de connaître l'auteur du message de bienvenue (utilisation d'un paramètre d'en-tête).

- Ajouter une nouvelle méthode Java getHello dans la classe HelloResource qui prend en paramètre une chaîne de caractères initialisée par un *template parameter* (annotations @Path et @PathParam) et une autre chaîne de caractères initialisée par un paramètre d'en-tête (annotation @HeaderParam) dont la clé sera name. La valeur par défaut de l'en-tête sera fixée votre serviteur (annotation @DefaultValue).

```
1.      @GET
2.      @Path("{id}")
3.      public String getHello(@PathParam("id") String id,
4.                             @DefaultValue("votre serviteur") @HeaderParam("name") String
 name) {
5.          return "Bonjour " + id + " de la part de " + name;
6.      }
```

- Exécuter de nouveau la classe HelloLauncher et depuis votre navigateur web saisir l'URL permettant d'invoquer ce nouveau service web REST (requête GET via l'URL **http://localhost:9991/api/hello/ENSMA**). Malheureusement, le navigateur web ne permet pas de préciser la valeur du paramètre d'en-tête name. Nous utiliserons donc l'outil en ligne de commande **cURL** pour construire des requêtes HTTP complexes.
- Depuis une invite de commande saisir la commande suivante :

```
1. $ curl --header "name:Mickael BARON" http://localhost:9991/api/hello/ENSMA
2. Bonjour ENSMA de la part de Mickael BARON
```

Ce service web REST n'est toujours pas complet, puisque nous aimerions retourner dans l'en-tête de la réponse, l'auteur du message de bienvenue. Comment pourrions-nous retourner à la fois un contenu dans la réponse et une information dans l'en-tête de la réponse ? Pour cela, nous allons utiliser un objet Response pour le retour de méthode.

- Ajouter une nouvelle méthode Java getHelloWithHeaders dans la classe HelloENSMA qui possède les mêmes paramètres que la précédente méthode. Le chemin pour invoquer cette méthode sera withheaders/{id} où id est le paramètre du message de bienvenue. Dans le corps de la méthode getHelloWithHeaders, compléter le code ci-dessous afin de transmettre le nom de l'auteur dans l'en-tête de la réponse.

```
1.      @GET
2.      @Path("withheaders/{id}")
3.      public Response getHelloWithHeaders(@PathParam("id") String id,
4.                                          @DefaultValue("votre serviteur") @HeaderParam("name")
 String name) {
5.          return Response.ok().header("name", name).entity("Bonjour " + id + " de la part de
 (voir l'en-tête).").build();
6.      }
```

- Exécuter de nouveau la classe HelloLauncher, puis saisir la ligne de commande **cURL** suivante pour envoyer une requête avec les bons paramètres et détailler le retour de la réponse.

```
1. $ curl --header "name:Mickael BARON" http://localhost:9991/api/hello/withheaders/ENSMA -v
2. *   Trying ::1...
3. * TCP_NODELAY set
4. * Connection failed
5. * connect to ::1 port 9991 failed: Connection refused
6. *   Trying 127.0.0.1...
7. * TCP_NODELAY set
8. * Connected to localhost (127.0.0.1) port 9991 (#0)
9. > GET /api/hello/withheaders/ENSMA HTTP/1.1
10. > Host: localhost:9991
11. > User-Agent: curl/7.54.0
12. > Accept: */*
13. > name:Mickael BARON
14. >
15. < HTTP/1.1 200 OK
16. < name: Mickael BARON
17. < Content-Type: text/plain
18. < Content-Length: 46
19. <
20. * Connection #0 to host localhost left intact
21. Bonjour ENSMA de la part de (voir l'en-tête).
```

- Nous allons afficher le contrat de description de ce service web REST au format WADL. Saisir depuis un navigateur l'URL suivante : **http://localhost:9991/api/application.wadl**.

```
1. <?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
2. <application xmlns="http://wadl.dev.java.net/2009/02">
3.     <doc xmlns:jersey="http://jersey.java.net/" jersey:generatedBy="Jersey: 2.27 2018-04-10
 07:34:57"/>
4.     <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:9991/api/application.wadl?detail=true"/>
5.     <grammars/>
6.     <resources base="http://localhost:9991/api/">
7.         <resource path="hello">
8.             <method id="getHello" name="GET">
9.                 <response>
10.                    <representation mediaType="text/plain"/>
11.                </response>
12.            </method>
13.            <resource path="{id}">
14.                <param xmlns:xs="http://www.w3.org/2001/
XMLSchema" name="id" style="template" type="xs:string"/>
15.                <method id="getHello" name="GET">
16.                    <request>
17.                        <param xmlns:xs="http://www.w3.org/2001/
XMLSchema" name="name" style="header" type="xs:string" default="votre serviteur"/>
18.                    </request>
19.                    <response>
20.                        <representation mediaType="text/plain"/>
21.                    </response>
22.                </method>
23.            </resource>
24.            <resource path="withheaders/{id}">
25.                <param xmlns:xs="http://www.w3.org/2001/
XMLSchema" name="id" style="template" type="xs:string"/>
26.                <method id="getHelloWithHeaders" name="GET">
27.                    <request>
28.                        <param xmlns:xs="http://www.w3.org/2001/
XMLSchema" name="name" style="header" type="xs:string" default="votre serviteur"/>
29.                    </request>
30.                    <response>
31.                        <representation mediaType="text/plain"/>
32.                    </response>
33.                </method>
34.            </resource>
35.        </resource>
36.    </resources>
37. </application>
```

II - Exercice 2 : développer un service web REST « Interrogation et réservation de billets de train »

II-A - But

- Développer un service web REST à partir de classes Java.
- Utiliser un *Sub-Resource Locator*.
- Manipuler des types personnalisés.
- Manipuler des formats de représentations (XML et JSON).

II-B - Description

Le service web REST de ce deuxième exercice consiste à créer un système CRUD pour l'interrogation et la réservation de billets de train. Les ressources manipulées par ce service web REST sont donc un **train** et une **réservation de billets de train** pour un train donné. Le service web REST doit pouvoir lister l'ensemble des trains, lister les trains qui satisfont un critère de recherche (ville de départ, ville d'arrivée, jour de départ et un intervalle de temps), puis de créer, lister et supprimer une réservation de billets de train.

Nous insisterons sur la mise en place du service web REST et non sur le code métier (le code Java dans le corps des méthodes est « sans importance »). Les formats supportés par les deux ressources seront du XML et du JSON.

II-C - Étapes à suivre

- Démarrer l'environnement de développement Eclipse.
- Importer le projet Maven **jaxrs-tutorial-exercice2** (**File -> Import -> Maven -> Existing Maven Projects**), choisir le répertoire du projet, puis faire **Finish**.

Le projet importé contient déjà des classes. À ce stade, de nombreuses erreurs de compilation sont présentes, dues à l'absence de certaines classes Java. Pas d'inquiétude, nous allons les ajouter progressivement.

- Créer une classe Train (dans le package fr.mickaelbaron.jaxrstutorialexercice2) qui modélise le concept de **train** et qui contient un attribut String id (identifiant fonctionnel d'un train), un attribut String departure (la ville de départ du train), un attribut String arrival (la ville d'arrivée du train) et un attribut int departureTime (heure de départ). Ajouter des modificateurs et des accesseurs sur tous les attributs. Ci-dessous un résultat du code que vous devez obtenir.

```
1. package fr.mickaelbaron.jaxrstutorialexercice2;
2.
3. public class Train {
4.
5.     private String id;
6.
7.     private String departure;
8.
9.     private String arrival;
10.
11.     private int departureTime; // Format : 1230 = 12h30
12.
13.     public Train() {
14.     }
15.
16.     public Train(String pId, String departure, String arrival, int departureTime) {
17.         this.id = pId;
18.         this.departure = departure;
19.         this.arrival = arrival;
20.         this.departureTime = departureTime;
21.     }
22.
23.     public String getId() {
24.         return id;
25.     }
26.
27.     public void setId(String pId) {
28.         this.id = pId;
29.     }
30.
31.     public String getDeparture() {
32.         return departure;
33.     }
34.
35.     public void setDeparture(String departure) {
36.         this.departure = departure;
37.     }
38.
39.     public String getArrival() {
40.         return arrival;
41.     }
42.
43.     public void setArrival(String arrival) {
44.         this.arrival = arrival;
45.     }
46.
47.     public int getDepartureTime() {
48.         return departureTime;
49.     }
50.
51.     public void setDepartureTime(int departureTime) {
```

```
52.         this.departureTime = departureTime;
53.     }
54. }
```

- Examiner la classe TrainBookingBD qui joue le rôle de DAO et de base de données. En effet, toutes les instances créées se feront en mémoire.
- Créer la classe TrainResource (dans le package fr.mickaelbaron.jaxrstutorialexercice2) permettant de représenter la ressource **train**. La classe contient trois méthodes qui permettent respectivement : 1) de retourner une liste de tous les trains ; 2) d'obtenir un train par son identifiant ; 3) de rechercher un train par des critères passés en paramètres (ville de départ, ville d'arrivée et heure de départ). Noter que les formats de retour peuvent être du XML ou du JSON. Compléter la classe ci-dessous en remplaçant TODO par les bonnes instructions JAX-RS.

```
1. package fr.mickaelbaron.jaxrstutorialexercice2;
2.
3. // TODO: le chemin racine de la ressource doit débuter par `/trains`.
4. // TODO: les formats du contenu des réponses sont XML puis JSON.
5. public class TrainResource {
6.
7.     public TrainResource() { }
8.
9.     // TODO: préciser le verbe HTTP.
10.     public List<Train> getTrains() {
11.         System.out.println("getTrains");
12.
13.         return TrainBookingDB.getTrains();
14.     }
15.
16.     // TODO: préciser le verbe HTTP
17.     // TODO: le chemin doit commencer par `/trainid-`  et se finir
18.     // par un template paramètre désignant l'identifiant du train.
19.     public Train getTrain(String trainId) {
20.         System.out.println("getTrain");
21.
22.         Optional<Train> trainById = TrainBookingDB.getTrainById(trainId);
23.
24.         if (trainById.isPresent()) {
25.             return trainById.get();
26.         } else {
27.             // TODO: déclencher une exception avec un statut NOT_FOUND.
28.         }
29.     }
30.
31.     // TODO: préciser le verbe HTTP.
32.     // TODO: le chemin doit commencer par `/#search`.
33.     // Les paramètres sont tous des paramètres de requête.
34.     public ??? searchTrainsByCriteria(String departure, String arrival, String
 departureTime) {
35.         System.out.println("TrainResource.searchTrainsByCriteria()");
36.
37.         // TODO: retourner une réponse avec :
38.         //   1/ les trois paramètres de requête en en-tête
39.         //   2/ un sous-ensemble de la liste des trains
40.         //      (exemple : `TrainBookingDB.getTrains().subList(0, 2)`)
41.     }
42. }
```

```
1. package fr.mickaelbaron.jaxrstutorialexercice2;
2.
3. @Path("/trains")
4. @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON })
5. public class TrainResource {
6.
7.     public TrainResource() { }
8.
9.     @GET
10.     public List<Train> getTrains() {
11.         System.out.println("getTrains");
```

```
12.
13.         return TrainBookingDB.getTrains();
14.     }
15.
16.     @GET
17.     @Path("trainid-{id}")
18.     public Train getTrain(@PathParam("id") String trainId) {
19.         System.out.println("getTrain");
20.
21.         Optional<Train> trainById = TrainBookingDB.getTrainById(trainId);
22.
23.         if (trainById.isPresent()) {
24.             return trainById.get();
25.         } else {
26.             throw new NotFoundException();
27.         }
28.     }
29.
30.     @GET
31.     @Path("/search")
32.     public Response searchTrainsByCriteria(@QueryParam("departure") String departure,
33.         @QueryParam("arrival") String arrival, @QueryParam("arrivalhour") String
 arrivalHour) {
34.         System.out.println("TrainResource.searchTrainsByCriteria()");
35.
36.         return Response.ok().header("departure", departure).header("arrival",
 arrival).header("arrivalhour", arrivalHour)
37.                 .entity(TrainBookingDB.getTrains().subList(0, 2)).build();
38.     }
39.
40.     @Path("/bookings")
41.     public TrainBookingResource getTrainBookingResource() {
42.         System.out.println("TrainResource.getTrainBookingResource()");
43.
44.         return new TrainBookingResource();
45.     }
46. }
```

- Créer la classe TrainBookingLauncher utilisée pour tester ce service web REST.

```
1. package fr.mickaelbaron.jaxrstutorialexercice2;
2.
3. public class TrainBookingLauncher {
4.
5.     public static final URI BASE_URI = getBaseURI();
6.
7.     private static URI getBaseURI() {
8.         return UriBuilder.fromUri("http://localhost/api/").port(9992).build();
9.     }
10.
11.     public static void main(String[] args) {
12.         ResourceConfig rc = new ResourceConfig();
13.         rc.registerClasses(TrainResource.class);
14.         rc.property(LoggingFeature.LOGGING_FEATURE_LOGGER_LEVEL_SERVER,
 Level.WARNING.getName());
15.
16.         try {
17.             HttpServer server = GrizzlyHttpServerFactory.createHttpServer(BASE_URI, rc);
18.             server.start();
19.
20.             System.out.println(String.format(
21.                 "Jersey app started with WADL available at " + "%sapplication.wadl\nHit enter
 to stop it...", BASE_URI, BASE_URI));
22.
23.             System.in.read();
24.             server.shutdownNow();
25.         } catch (Exception e) {
26.             e.printStackTrace();
27.         }
28.     }
```

```
29. }
```

- Exécuter la classe TrainBookingLauncher, puis à partir de **cURL** invoquer le service web permettant de récupérer la liste de tous les trains.

```
1. $ curl http://localhost:9992/api/trains -v
2. * Connected to localhost (127.0.0.1) port 9992 (#0)
3. > GET /api/trains HTTP/1.1
4. > Host: localhost:9992
5. > User-Agent: curl/7.54.0
6. > Accept: */*
7. >
8. < HTTP/1.1 500 Internal Server Error
9. < Connection: close
10. < Content-Length: 0
11. <
12. * Closing connection 0
```

Comme observé sur le retour de la commande **cURL**, une erreur 500 (*Internal Server Error*) est retournée. Sur la console du serveur, le message suivant a dû être généré.

```
1. nov. 06, 2018 6:29:44 PM org.glassfish.jersey.message.internal.WriterInterceptorExecutor
$TerminalWriterInterceptor aroundWriteTo
2. SEVERE: MessageBodyWriter not found for media type=application/xml, type=class
 java.util.ArrayList, genericType=java.util.List<fr.mickaelbaron.jaxrstutorialexercice2.Train>.
```

Cette erreur indique que la classe Train ne contient pas les informations nécessaires pour transformer un objet en XML (XML est le format choisi dans l'ordre d'apparition de l'annotation @Produces). Pour une sérialisation Java <=> XML, chaque classe doit être au moins annotée à la racine pour activer le *mapping* entre le XML Schema et les attributs de la classe.

- Éditer la classe Train et ajouter l'annotation suivante.

```
1. @XmlRootElement(name = "train")
2. public class Train {
3.     ...
4. }
```

- Exécuter de nouveau la classe TrainBookingLauncher, puis à partir de **cURL** invoquer le service web permettant de récupérer la liste de tous les trains. Comme montré ci-dessous, la liste des trains est obtenue via un format XML.

```
1. $ curl http://localhost:9992/api/trains
2. <?xml version="1.0" encoding="UTF-8" standalone="yes"?><trains><train><arrival>Paris</
arrival><departure>Poitiers</departure><departureTime>1250</departureTime><id>TR123</id></
train><train><arrival>Paris</arrival><departure>Poitiers</departure><departureTime>1420</
departureTime><id>AX127</id></train><train><arrival>Paris</arrival><departure>Poitiers</
departure><departureTime>1710</departureTime><id>PT911</id></train></trains>
```

Nous souhaitons désormais obtenir un retour du contenu au format JSON. Cette information doit être précisée dans l'en-tête de la requête avec l'attribut Accept.

- Saisir la ligne de commande suivante.

```
1. $ curl --header "Accept:application/json" http://localhost:9992/api/trains -v
2. * Connected to localhost (127.0.0.1) port 9992 (#0)
3. > GET /api/trains HTTP/1.1
4. > Host: localhost:9992
5. > User-Agent: curl/7.54.0
6. > Accept: application/json
7. >
8. < HTTP/1.1 500 Internal Server Error
```

```
9. < Connection: close
10. < Content-Length: 0
11. <
12. * Closing connection 0
```

Comme observé sur le retour de la commande **cURL**, une erreur 500 est retournée. Sur la console du serveur, le message est identique à la précédente erreur pour le format XML. En d'autres termes, Jersey ne sait pas comment transformer un objet Java en JSON. Pour résoudre cette absence, il suffit d'ajouter la dépendance de la bibliothèque Jackson au fichier *pom.xml*.

```
1. nov. 06, 2018 7:30:14 PM org.glassfish.jersey.message.internal.WriterInterceptorExecutor
$TerminalWriterInterceptor aroundWriteTo
2. SEVERE: MessageBodyWriter not found for media type=application/json, type=class
 java.util.ArrayList, genericType=java.util.List<fr.mickaelbaron.jaxrstutorialexercice2.Train>.
```

- Éditer le fichier _pom.xml_ et ajouter la dépendance suivante.

```
1. <dependency>
2.     <groupId>org.glassfish.jersey.media</groupId>
3.     <artifactId>jersey-media-json-jackson</artifactId>
4. </dependency>
```

- Exécuter de nouveau la classe TrainBookingLauncher, puis exécuter de nouveau la commande **cURL** précédente.

```
1. $ curl --header "Accept: application/json" http://localhost:9992/api/trains
2. [{"id":"TR123","departure":"Poitiers","arrival":"Paris","departureTime":1250},
{"id":"AX127","departure":"Poitiers","arrival":"Paris","departureTime":1420},
{"id":"PT911","departure":"Poitiers","arrival":"Paris","departureTime":1710}]
```

Vous constatez sur le résultat JSON que le nom des clés correspond exactement au nom des attributs de la classe Java Train.

- Modifier la classe Train afin d'obtenir une clé departure_time (dans le format JSON), au lieu de departureTime, tout en respectant les conventions de nommage Java.

```
1. @XmlRootElement(name = "train")
2. public class Train {
3.
4.     private String id;
5.
6.     private String departure;
7.
8.     private String arrival;
9.
10.     @JsonProperty("departure_time")
11.     private int departureTime; // Format : 1230 = 12h30
12.     ...
13. }
```

- Exécuter la classe TrainBookingLauncher, puis vérifier que le nom de la clé departure_time a été impacté.

```
1. $ curl --header "Accept: application/json" http://localhost:9992/api/trains
2. [{"id":"TR123","departure":"Poitiers","arrival":"Paris","departure_time":1250},
{"id":"AX127","departure":"Poitiers","arrival":"Paris","departure_time":1420},
{"id":"PT911","departure":"Poitiers","arrival":"Paris","departure_time":1710}]
```

- Continuer à tester le service web REST de façon à invoquer les méthodes Java getTrain et searchTrainsByCriteria. Pour cette dernière méthode, afficher la réponse complète pour s'assurer que les trois paramètres de requête sont transmis dans l'en-tête de la réponse.

```
1. $ curl --header "Accept: application/json" http://localhost:9992/api/trains/trainid-TR123
```

```
2. {"id":"TR123","departure":"Poitiers","arrival":"Paris","departure_time":1250}
3.
4. $ curl --header "Accept: application/json" http://localhost:9992/api/trains/search\?departure
\=poitiers\&arrival\=paris\&departure_time\=1050 -v
5. * Connected to localhost (127.0.0.1) port 9992 (#0)
6. > GET /api/trains/search?departure=poitiers&arrival=paris&departure_time=1050 HTTP/1.1
7. > Host: localhost:9992
8. > User-Agent: curl/7.54.0
9. > Accept: application/json
10. >
11. < HTTP/1.1 200 OK
12. < departure: poitiers
13. < arrival: paris
14. < arrivalhour: 1050
15. < Content-Type: application/json
16. < Content-Length: 157
17. <
18. * Connection #0 to host localhost left intact
19. [{"id":"TR123","departure":"Poitiers","arrival":"Paris","departure_time":1250},
{"id":"AX127","departure":"Poitiers","arrival":"Paris","departure_time":1420}]
```

Nous allons maintenant nous occuper à implémenter le service dédié à la **réservation de billets de train** pour un train donné. La classe TrainBooking utilisée pour modéliser une réservation de billets est déjà présente dans le projet. Elle contient un attribut String id (identifiant fonctionnel d'une réservation de billets), un attribut String trainId (la clé étrangère du train) et un attribut int numberPlaces pour le nombre de places à réserver. Toutefois, le paramétrage de la classe BookTrainResource est utilisé pour implémenter le service de réservation de billets.

- Modifier la classe TrainBooking afin d'obtenir une clé current_train (dans le format JSON) au lieu de trainId et une clé number_places (dans le format JSON) au lieu de numberPlaces.

```
1. @XmlRootElement(name = "trainbooking")
2. public class TrainBooking {
3.
4.     private String id;
5.
6.     // TODO: le nom de la clé JSON doit être current_train.
7.     private String trainId;
8.
9.     // TODO: le nom de la clé JSON doit être number_places.
10.     private int numberPlaces;
11.     ...
12. }
```

```
1. @XmlRootElement(name = "trainbooking")
2. public class TrainBooking {
3.
4.     private String id;
5.
6.     @JsonProperty("current_train")
7.     private String trainId;
8.
9.     @JsonProperty("number_places")
10.     private int numberPlaces;
11.     ...
12. }
```

- Créer la classe BookTrainResource (dans le package fr.mickaelbaron.jaxrstutorialexercice2). Quatre méthodes sont à définir. La première createTrainBooking est invoquée pour la création d'une réservation de billets. La deuxième getTrainBookings est utilisée pour lister l'ensemble des réservations. La troisième getTrainBooking permet de retourner les informations d'une réservation à partir d'un numéro de réservation. Finalement removeBookTrain permet de supprimer une réservation.

```
1. package fr.mickaelbaron.jaxrstutorialexercice2;
2.
3. public class TrainBookingResource {
4.
```

```
5.      // TODO: préciser le verbe HTTP.
6.      public TrainBooking createTrainBooking(TrainBooking trainBooking) {
7.          System.out.println("TrainBookingResource.createTrainBooking()");
8.
9.          Optional<Train> findFirst = TrainBookingDB.getTrainById(trainBooking.getTrainId());
10.
11.          if (findFirst.isEmpty()) {
12.              // TODO: déclencher une exception avec un statut NOT_FOUND.
13.          }
14.
15.          TrainBooking newBookTrain = new TrainBooking();
16.          newBookTrain.setNumberPlaces(trainBooking.getNumberPlaces());
17.          newBookTrain.setTrainId(findFirst.get().getId());
18.          newBookTrain.setId(Long.toString(System.currentTimeMillis()));
19.
20.          TrainBookingDB.getTrainBookings().add(newBookTrain);
21.
22.          return newBookTrain;
23.      }
24.
25.      // TODO: préciser le verbe HTTP.
26.      public List<TrainBooking> getTrainBookings() {
27.          System.out.println("TrainBookingResource.getTrainBookings()");
28.
29.          return TrainBookingDB.getTrainBookings();
30.      }
31.
32.      // TODO: préciser le verbe HTTP.
33.      // TODO: template paramètre désignant l'identifiant du train.
34.      public TrainBooking getTrainBooking(String trainBookingId) {
35.          System.out.println("TrainBookingResource.getTrainBooking()");
36.
37.          Optional<TrainBooking> findFirst =
 TrainBookingDB.getTrainBookingById(trainBookingId);
38.
39.          if (findFirst.isPresent()) {
40.              return findFirst.get();
41.          } else {
42.              // TODO: déclencher une exception avec un statut NOT_FOUND.
43.          }
44.      }
45.
46.      // TODO: préciser le verbe HTTP.
47.      // TODO: template paramètre désignant l'identifiant du train.
48.      public void removeTrainBooking(@PathParam("id") String trainBookingId) {
49.          System.out.println("TrainBookingResource.removeTrainBooking()");
50.
51.          Optional<TrainBooking> findFirst =
 TrainBookingDB.getTrainBookingById(trainBookingId);
52.
53.          if (findFirst.isPresent()) {
54.              TrainBookingDB.getTrainBookings().remove(findFirst.get());
55.          } else {
56.              // TODO: déclencher une exception avec un statut NOT_FOUND.
57.          }
58.      }
59. }
```

```
1. package fr.mickaelbaron.jaxrstutorialexercice2;
2.
3. public class TrainBookingResource {
4.
5.      @POST
6.      public TrainBooking createTrainBooking(TrainBooking trainBooking) {
7.          System.out.println("TrainBookingResource.createTrainBooking()");
8.
9.          Optional<Train> findFirst = TrainBookingDB.getTrainById(trainBooking.getTrainId());
10.
11.          if (findFirst.isEmpty()) {
12.              throw new NotFoundException("No matches found.");
13.          }
```

```
14.
15.         TrainBooking newBookTrain = new TrainBooking();
16.         newBookTrain.setNumberPlaces(trainBooking.getNumberPlaces());
17.         newBookTrain.setTrainId(findFirst.get().getId());
18.         newBookTrain.setId(Long.toString(System.currentTimeMillis()));
19.
20.         TrainBookingDB.getTrainBookings().add(newBookTrain);
21.
22.         return newBookTrain;
23.     }
24.
25.     @GET
26.     public List<TrainBooking> getTrainBookings() {
27.         System.out.println("TrainBookingResource.getTrainBookings()");
28.
29.         return TrainBookingDB.getTrainBookings();
30.     }
31.
32.     @GET
33.     @Path("{id}")
34.     public TrainBooking getTrainBooking(String trainBookingId) {
35.         System.out.println("TrainBookingResource.getTrainBooking()");
36.
37.         Optional<TrainBooking> findFirst =
 TrainBookingDB.getTrainBookingById(trainBookingId);
38.
39.         if (findFirst.isPresent()) {
40.             return findFirst.get();
41.         } else {
42.             throw new NotFoundException("No matches found.");
43.         }
44.     }
45.
46.     @DELETE
47.     @Path("{id}")
48.     public void removeTrainBooking(@PathParam("id") String trainBookingId) {
49.         System.out.println("TrainBookingResource.removeTrainBooking()");
50.
51.         Optional<TrainBooking> findFirst =
 TrainBookingDB.getTrainBookingById(trainBookingId);
52.
53.         if (findFirst.isPresent()) {
54.             TrainBookingDB.getTrainBookings().remove(findFirst.get());
55.         } else {
56.             throw new NotFoundException("No matches found.");
57.         }
58.     }
59. }
```

Cette ressource définie par la classe TrainBookingResource est accessible via la mise en place d'un *sub-resource locator* depuis la ressource *train*. L'avantage est de pouvoir lier la ressource *train* avec la ressource de *réservation de billets*.

- Compléter la classe TrainResource de façon à ajouter une méthode getTrainBookingResource qui servira de *sub-resource locator*. Assurer que les exigences suivantes soient respectées : annotée avec @Path, non annotée avec les annotations de méthodes HTTP (@GET, @POST, @PUT et @DELETE) et doit retourner un objet de type ressource.

```
1. ...
2. public class TrainResource {
3.     ...
4.
5.     // TODO: la sous-ressource doit être accessible via l'URI `/bookings`
6.     public ??? getTrainBookingResource() {
7.         System.out.println("TrainResource.getTrainBookingResource()");
8.
9.         // TODO: doit retourner un objet du type de la ressource souhaitée =>
 TrainBookingResource
10.     }
```

```
11. }
```

```
1. ...
2. public class TrainResource {
3.     ...
4.
5.     @Path("/bookings")
6.     public ??? getTrainBookingResource() {
7.         System.out.println("TrainResource.getTrainBookingResource()");
8.
9.         return new TrainBookingResource();
10.     }
11. }
```

- Exécuter la classe TrainBookingLauncher et à partir de **cURL** invoquer chaque service lié à la réservation de billets de train qui ont été implémentés dans les quatre méthodes createTrainBooking, getTrainBookings, getTrainBooking et removeTrainBooking.

```
1. # Récupérer la liste des trains.
2. $ curl --header "Accept: application/json" http://localhost:9992/api/trains
3. [{"id":"TR123","departure":"Poitiers","arrival":"Paris","departure_time":1250},
{"id":"AX127","departure":"Poitiers","arrival":"Paris","departure_time":1420},
{"id":"PT911","departure":"Poitiers","arrival":"Paris","departure_time":1710}]
4.
5. # Créer une réservation de billets de train.
6. $ curl --header "Accept: application/json" --header "Content-Type: application/json" --request
 POST --data '{"current_train":"TR123","number_places":2}' http://localhost:9992/api/trains/
bookings
7. {"id":"1541683057395","current_train":"TR123","number_places":2}
8.
9. # Récupérer la liste des réservations de billets de train.
10. $ curl --header "Accept: application/json" http://localhost:9992/api/trains/bookings
11. [{"id":"1541683057395","current_train":"TR123","number_places":2}]
12.
13. # Récupérer une réservation de billets de train par un identifiant.
14. $ curl --header "Accept: application/json" http://localhost:9992/api/trains/
bookings/1541683057395
15. [{"id":"1541683057395","current_train":"TR123","number_places":2}]
16.
17. # Supprimer une réservation de billets de train par un identifiant (réussie).
18. $ curl --request DELETE http://localhost:9992/api/trains/bookings/1541683057395 -v
19. * Connected to localhost (127.0.0.1) port 9992 (#0)
20. > DELETE /api/trains/bookings/1541685562466 HTTP/1.1
21. > Host: localhost:9992
22. > User-Agent: curl/7.54.0
23. > Accept: */*
24. >
25. < HTTP/1.1 204 No Content
26. <
27. * Connection #0 to host localhost left intact
28.
29. # Comme la méthode DELETE est idempotente possibilité de rappeler plusieurs fois la
 suppression de billets de train pour un même identifiant (réussie).
30. $ curl --request DELETE http://localhost:9992/api/trains/bookings/1541683057395 -v
31. * Connected to localhost (127.0.0.1) port 9992 (#0)
32. > DELETE /api/trains/bookings/1541685562466 HTTP/1.1
33. > Host: localhost:9992
34. > User-Agent: curl/7.54.0
35. > Accept: */*
36. >
37. < HTTP/1.1 204 No Content
38. <
39. * Connection #0 to host localhost left intact
```

III - Exercice 3 : tests d'intégration de service web REST « Interrogation et réservation de billets de train »

III-A - But

- Utiliser le framework de test **Test-Framework** de Jersey.
- Utiliser un style de développement de tests basé sur *Given-When-Then*.
- Savoir écrire des tests d'intégration.

III-B - Description

Ce troisième exercice s'intéresse aux tests d'intégration de service web REST développés avec JAX-RS et Jersey. Le code du service web REST est déjà fourni et correspond à la solution complète de l'exercice 2 sur l'interrogation et la réservation de billets de train. Nous insistons dans cet exercice sur les bonnes pratiques pour développer des tests des services web REST en s'appuyant sur le style *Given-When-Then*.

Dans la suite de cet exercice, le format de représentation des objets sera du JSON. Pour chaque méthode de test implémentée, vous exécuterez le test unitaire associé.

III-C - Étapes à suivre

- Démarrer l'environnement de développement Eclipse.
- Importer le projet Maven **jaxrs-tutorial-exercice3** (**File -> Import -> Maven -> Existing Maven Projects**), choisir le répertoire du projet, puis faire **Finish**.

Le projet importé contient déjà une implémentation complète du service web REST dédié à l'interrogation et la réservation de billets de train.

- Éditer le fichier de description Maven *pom.xml* et ajouter les dépendances suivantes afin d'utiliser le framework de test Test-Framework de Jersey.

```
1. <dependency>
2.     <groupId>org.glassfish.jersey.test-framework</groupId>
3.     <artifactId>jersey-test-framework-core</artifactId>
4.     <scope>test</scope>
5. </dependency>
6. <dependency>
7.     <groupId>org.glassfish.jersey.test-framework.providers</groupId>
8.     <artifactId>jersey-test-framework-provider-grizzly2</artifactId>
9.     <scope>test</scope>
10. </dependency>
```

- Ouvrir la classe TrainResourceIntegrationTest et compléter le code afin que le framework de test **Test-Framework** soit supporté.

```
1. public class TrainResourceIntegrationTest extends JerseyTest {
2.
3.     @Override
4.     protected Application configure() {
5.         ResourceConfig resourceConfig = new ResourceConfig(TrainResource.class);
6.         resourceConfig.property(LoggingFeature.LOGGING_FEATURE_LOGGER_LEVEL_SERVER,
 Level.WARNING.getName());
7.         return resourceConfig;
8.     }
```

- Ajouter la méthode getTrainsTest qui permet de tester le comportement du service dédié à la récupération de tous les trains /trains.

```
1.      @Test
2.      public void getTrainsTest() {
3.          // Given
4.
5.          // When
6.          Response response = target("/trains").request(MediaType.APPLICATION_JSON_TYPE).get();
7.
8.          // Then
9.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
10.          List<Train> readEntities = response.readEntity(new GenericType<List<Train>>() {});
11.          Assert.assertNotNull(readEntities);
12.          Assert.assertEquals(3, readEntities.size());
13.          Assert.assertTrue(readEntities.stream().anyMatch(current -
> "TR123".equals(current.getId())));
14.      }
```

- Compléter la méthode getTrainTest au niveau de la partie *When* afin de satisfaire les différentes assertions.

```
1.      @Test
2.      public void getTrainTest() {
3.          // Given
4.          String trainId = "TR123";
5.
6.          // When
7.          // TODO: invoquer le service dédié à la récupération d'un train
8.          // par son identifiant fonctionnel (trainid = "TR123").
9.
10.          // Then
11.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
12.          Assert.assertEquals(MediaType.APPLICATION_JSON_TYPE, response.getMediaType());
13.          Train readEntity = response.readEntity(Train.class);
14.          Assert.assertNotNull(readEntity);
15.          Assert.assertEquals("Poitiers", readEntity.getDeparture());
16.      }
```

```
1.      @Test
2.      public void getTrainTest() {
3.          // Given
4.          String trainId = "TR123";
5.
6.          // When
7.          Response response = target("/trains").path("trainid-" +
 trainId).request(MediaType.APPLICATION_JSON_TYPE).get();
8.
9.          // Then
10.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
11.          Assert.assertEquals(MediaType.APPLICATION_JSON_TYPE, response.getMediaType());
12.          Train readEntity = response.readEntity(Train.class);
13.          Assert.assertNotNull(readEntity);
14.          Assert.assertEquals("Poitiers", readEntity.getDeparture());
15.      }
```

- Compléter la méthode searchTrainsByCriteriaTest au niveau de la partie *Then* afin d'ajouter des assertions qui satisfont les contraintes suivantes : le code de statut doit être 200, la réponse doit contenir trois paramètres d'en-tête qui correspondent aux paramètres de la requête initiale (departure, arrival et departure_time) et le contenu doit être une liste de trains d'une taille de deux éléments.

```
1.      @Test
2.      public void searchTrainsByCriteriaTest() {
3.          // Given
4.          String departure = "Poitiers";
5.          String arrival = "Paris";
6.          String departureTime = "1710";
7.
8.          // When
```

```
9.          Response response = target("/trains").path("search").queryParam("departure",
 departure)
10.                 .queryParam("arrival", arrival).queryParam("departure_time", departureTime)
11.                 .request(MediaType.APPLICATION_JSON_TYPE).get();
12.
13.         // Then
14.         // TODO: assertions à respecter ?
15.         //  * le code de statut doit être `200` ;
16.         //  * la réponse doit contenir trois paramètres d'en-tête qui correspondent
17.         //    aux paramètres de la requête initiale (`departure`, `arrival` et
 `departure_time`) ;
18.         //  * le contenu doit être une liste de trains d'une taille de deux éléments.
19.     }
```

```
1.      @Test
2.      public void searchTrainsByCriteriaTest() {
3.          // Given
4.          String departure = "Poitiers";
5.          String arrival = "Paris";
6.          String departureTime = "1710";
7.
8.          // When
9.          Response response = target("/trains").path("search").queryParam("departure",
 departure)
10.                 .queryParam("arrival", arrival).queryParam("departureTime", departureTime)
11.                 .request(MediaType.APPLICATION_JSON_TYPE).get();
12.
13.         // Then
14.         Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
15.         Assert.assertEquals(departure, response.getHeaderString("departure"));
16.         Assert.assertEquals(arrival, response.getHeaderString("arrival"));
17.         Assert.assertEquals(departureTime, response.getHeaderString("departureTime"));
18.         List<Train> readEntities = response.readEntity(new GenericType<List<Train>>() {});
19.         Assert.assertNotNull(readEntities);
20.         Assert.assertEquals(2, readEntities.size());
21.     }
```

Nous allons maintenant implémenter les tests d'intégration concernant la ressource de *réservation de billets de train*.

- Ouvrir la classe TrainBookingResourceIntegrationTest et compléter le code afin que le framework de test **Test-Framework** soit supporté.

```
1. public class TrainBookingResourceIntegrationTest extends ??? {
2.
3.      @Override
4.      protected Application configure() {
5.          // TODO: prendre en compte `TrainResource` et `TrainBookingResource`.
6.      }
```

```
1. public class TrainBookingResourceIntegrationTest extends JerseyTest {
2.
3.      @Override
4.      protected Application configure() {
5.          ResourceConfig resourceConfig = new ResourceConfig(TrainResource.class,
 TrainBookingResource.class);
6.          resourceConfig.property(LoggingFeature.LOGGING_FEATURE_LOGGER_LEVEL_SERVER,
 Level.WARNING.getName());
7.          return resourceConfig;
8.      }
```

- Ajouter une méthode createTrainBookingTest qui permet de tester le comportement du service dédié à la création des réservations de billets de train.

```
1.      @Test
2.      public void createTrainBookingTest() {
3.          // Given
```

```
4.          TrainBooking trainBooking = new TrainBooking();
5.          trainBooking.setNumberPlaces(3);
6.          trainBooking.setTrainId("TR123");
7.
8.          // When
9.          // TODO: invoquer le service web pour la création des réservations de billets de
 train.
10.
11.          // Then
12.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
13.      }
```

```
1.     @Test
2.      public void createTrainBookingTest() {
3.          // Given
4.          TrainBooking trainBooking = new TrainBooking();
5.          trainBooking.setNumberPlaces(3);
6.          trainBooking.setTrainId("TR123");
7.
8.          // When
9.          Response response = target("/trains/
bookings").request(MediaType.APPLICATION_JSON_TYPE)
10.                 .post(Entity.entity(trainBooking, MediaType.APPLICATION_JSON));
11.
12.          // Then
13.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
14.      }
```

- Ajouter une méthode createTrainBookingWithBadTrainIdTest qui permet de tester que si un identifiant de train n'existe pas, une erreur de type 404 est retournée (NOT_FOUND).

```
1.      @Test
2.      public void createTrainBookingWithBadTrainIdTest() {
3.          // Given
4.          TrainBooking trainBooking = new TrainBooking();
5.          trainBooking.setNumberPlaces(3);
6.          trainBooking.setTrainId("BADTR123");
7.
8.          // When
9.          // TODO: invoquer le service web pour la création des réservations de billets de
 train.
10.
11.          // Then
12.          Assert.assertEquals("Http Response should be 404: ",
 Status.NOT_FOUND.getStatusCode(), response.getStatus());
13.      }
```

```
1.      @Test
2.      public void createTrainBookingWithBadTrainIdTest() {
3.          // Given
4.          TrainBooking trainBooking = new TrainBooking();
5.          trainBooking.setNumberPlaces(3);
6.          trainBooking.setTrainId("BADTR123");
7.
8.          // When
9.          Response response = target("/trains/
bookings").request(MediaType.APPLICATION_JSON_TYPE)
10.                 .post(Entity.entity(trainBooking, MediaType.APPLICATION_JSON));
11.
12.          // Then
13.          Assert.assertEquals("Http Response should be 404: ",
 Status.NOT_FOUND.getStatusCode(), response.getStatus());
14.      }
```

- Ajouter une méthode getTrainBookingsTest qui permet de tester la récupération de toutes les réservations de train. Pour ce test, nous allons utiliser une fabrique de *réservations de billets de train* dans la partie Given.

```
1.      @Test
2.      public void getTrainBookingsTest() {
3.          // Given
4.          TrainBooking currentTrainBooking = createTrainBooking("TR123", 3);
5.
6.          // When
7.          // TODO: invoquer le service web pour la récupérer de toutes
8.          // les réservations de billets de train.
9.
10.          // Then
11.          // TODO: assertions à respecter ?
12.          //  * le code statut doit être `200` ;
13.          //  * une seule réservation de billets de train ;
14.          //  * l'identifiant du train passé lors de la création
15.          //    est le même que celui transmis par la réponse.
16.      }
```

```
1.      @Test
2.      public void getTrainBookingsTest() {
3.          // Given
4.          TrainBooking currentTrainBooking = createTrainBooking("TR123", 3);
5.
6.          // When
7.          Response response = target("/trains/
bookings").request(MediaType.APPLICATION_JSON_TYPE).get();
8.
9.          // Then
10.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
11.          List<TrainBooking> trainBookings = response.readEntity(new
 GenericType<List<TrainBooking>>() {});
12.          Assert.assertEquals(1, trainBookings.size());
13.          Assert.assertEquals(currentTrainBooking.getTrainId(),
 trainBookings.get(0).getTrainId());
14.      }
```

- Ajouter une méthode `getTrainBookingTest` qui permet de tester la récupération d'une réservation de billets de train à partir d'un identifiant de réservation.

```
1.      @Test
2.      public void getTrainBookingTest() {
3.          // Given
4.          TrainBooking currentTrainBooking = createTrainBooking("TR123", 3);
5.
6.          // When
7.          // TODO: invoquer le service web pour la récupération
8.          // d'une réservation de billets de train à partir de currentTrainBooking.getId().
9.
10.          // Then
11.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
12.      }
```

```
1.      @Test
2.      public void getTrainBookingTest() {
3.          // Given
4.          TrainBooking currentTrainBooking = createTrainBooking("TR123", 3);
5.
6.          // When
7.          Response response = target("/trains/
bookings").path(currentTrainBooking.getId()).request(MediaType.APPLICATION_JSON_TYPE).get();
8.
9.          // Then
10.          Assert.assertEquals("Http Response should be 200: ", Status.OK.getStatusCode(),
 response.getStatus());
11.      }
```

- Ajouter une méthode getTrainBookingWithBadTrainBookingIdTest qui permet de tester que si un identifiant de réservation de billets de train n'existe pas, le code statut de la réponse doit être 204 puisque la méthode DELETE est idempotente.

```
1.      @Test
2.      public void getTrainBookingWithBadTrainBookingIdTest() {
3.          // Given
4.          String trainBookingId = "FAKETRAINBOOKINGID";
5.
6.          // When
7.          // TODO: invoquer le service web pour la création des réservations de billets de
 train.
8.
9.          // Then
10.          // TODO: assertion doit vérifier que le code statut est `404`.
11.      }
```

```
1.      @Test
2.      public void getTrainBookingWithBadTrainBookingIdTest() {
3.          // Given
4.          String trainBookingId = "FAKETRAINBOOKINGID";
5.
6.          // When
7.          Response response = target("/trains/
bookings").path(trainBookingId).request(MediaType.APPLICATION_JSON_TYPE).get();
8.
9.          // Then
10.          Assert.assertEquals("Http Response should be 404: ",
 Status.NOT_FOUND.getStatusCode(), response.getStatus());
11.      }
```

- Ajouter une méthode removeTrainBookingTest qui permet de tester la suppression d'une réservation d'un billet de train. Le code statut de la réponse doit être 204.

```
1.      @Test
2.      public void removeTrainBookingTest() {
3.          // Given
4.          TrainBooking currentTrainBooking = createTrainBooking("TR123", 3);
5.
6.          // When
7.          // TODO: invoquer le service web pour la suppression d'une réservation de billets de
 train à partir de currentTrainBooking.getId()
8.
9.          // Then
10.          // TODO: assertion doit vérifier que le code statut est `204`.
11.      }
```

```
1.      @Test
2.      public void removeTrainBookingTest() {
3.          // Given
4.          TrainBooking currentTrainBooking = createTrainBooking("TR123", 3);
5.
6.          // When
7.          Response response = target("/trains/
bookings").path(currentTrainBooking.getId()).request(MediaType.APPLICATION_JSON_TYPE).delete();
8.
9.          // Then
10.          Assert.assertEquals("Http Response should be 204: ",
 Status.NO_CONTENT.getStatusCode(), response.getStatus());
11.      }
```

- Ajouter une méthode removeTrainBookingWithBadTrainBookingIdTest qui permet de tester que si un identifiant de réservation de billets de train n'existe pas, une erreur de type 404 est retournée (NOT_FOUND).

```
1.      @Test
2.      public void removeTrainBookingWithBadTrainBookingIdTest() {
3.          // Given
```

```
4.          String trainBookingId = "FAKETRAINBOOKINGID";
5.
6.          // When
7.          Response response = target("/trains/
bookings").path(trainBookingId).request(MediaType.APPLICATION_JSON_TYPE).delete();
8.
9.          // Then
10.          Assert.assertEquals("Http Response should be 404: ",
 Status.NOT_FOUND.getStatusCode(), response.getStatus());
11.      }
```

IV - Exercice 4 : client de service web REST « Interrogation et réservation de billets de train »

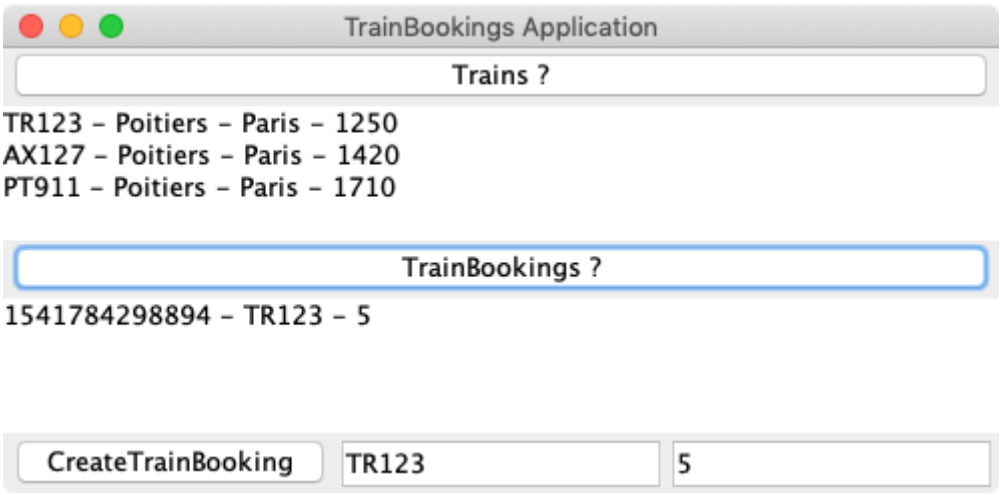
IV-A - But

- Utiliser l'API cliente de JAX-RS et son implémentation via Jersey.
- Savoir manipuler le type de contenu.

IV-B - Description

Ce quatrième exercice s'intéresse à la mise en place d'un client pour l'accès au service web REST développé dans les exercices 2 et 3. Une interface graphique développée en Java/Swing permet d'invoquer les services web pour les ressources de *train* et de *réservation de billets de train*.

Le projet de l'exercice 3 fournira l'implémentation du service web REST que le client que nous allons développer va invoquer. Exécuter la classe TrainBookingLauncher de l'exercice 3 pour rendre disponible le service web REST.



IV-C - Étapes à suivre

- Démarrer l'environnement de développement Eclipse.
- Importer le projet Maven **jaxrs-tutorial-exercice4** (**File -> Import -> Maven -> Existing Maven Projects**), choisir le répertoire du projet, puis faire **Finish**.

Le projet importé contient déjà une implémentation complète de l'interface graphique développée en Java/Swing. Aucune compétence en Java/Swing n'est demandée puisque l'objectif de cet exercice est de manipuler exclusivement l'API cliente JAX-RS. Si vous souhaitez proposer des améliorations à l'interface graphique, les *pull requests* sont les bienvenues.

- Éditer le fichier de description Maven *pom.xml* et ajouter la dépendance suivante afin d'utiliser l'API cliente JAX-RS et son implémentation Jersey.

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

- Ouvrir la classe TrainBookingsClientMain et compléter le code de la méthode initializeService afin d'initialiser l'attribut WebTarget target. Pour rappel, l'URL d'accès au service web REST est **http://localhost:9993/api/trains**.

```
    private void initializeService() {
        Client client = ClientBuilder.newClient();
        target = client.target("http://localhost:9993/api/trains");
    }
```

- Compléter la méthode callGetTrains permettant de récupérer l'ensemble des trains et de les afficher dans le composant graphique JTextArea.

```
    private void callGetTrains() {
        // TODO: invoquer le service web REST pour récupérer l'ensemble des trains
        // disponibles. Le résultat doit être transmis dans un objet de type
        // List<Train>.

        trainsConsole.setText("");
        for (Train current : result) {
            trainsConsole.append(current.getId() + " - " + current.getDeparture() + " - " +
 current.getArrival() + " - "
                + current.getDepartureTime() + "\n");
         }
     }
```

```
    private void callGetTrains() {
        List<Train> result = target.request(MediaType.APPLICATION_JSON_TYPE).get(new
 GenericType<List<Train>>() {
        });

        trainsConsole.setText("");
        for (Train current : result) {
            trainsConsole.append(current.getId() + " - " + current.getDeparture() + " - " +
 current.getArrival() + " - "
                + current.getDepartureTime() + "\n");
        }
    }
```

La classe Train n'existe pas dans le projet. Comme nous n'utilisons pas une démarche Top/Down, toutes les classes utilisées pour transmettre des données via les requêtes et les réponses ne sont pas générées automatiquement. La solution la plus rapide est de « copier/coller » la classe Train de l'exercice 3.

- Copier depuis l'exercice 3, la classe Train en vous assurant si possible de modifier le nom du package par fr.mickaelbaron.jaxrstutorialexercice4.

```
package fr.mickaelbaron.jaxrstutorialexercice5;

@XmlRootElement(name = "train")
public class Train {

    private String id;

    private String departure;

     private String arrival;

```

```
12.     @JsonProperty("departure_time")
13.     private int departureTime; // Format : 1230 = 12h30
14.
15.     ...
```

- Compléter la méthode createTrainBooking permettant de créer des réservations de billets de train. Les informations pour la création comme l'identifiant du train et le nombre de places sont renseignées en paramètre de la méthode (valeurs extraites depuis les champs de texte de l'interface graphique).

```
1.     private void createTrainBooking(String numTrain, int numberPlaces) {
2.         // TODO: créer un objet de type TrainBooking.
3.
4.         // TODO: invoquer le service web REST pour créer une réservation de train
5.         // à partir de l'objet TrainBooking.
6.
7.         if (Status.OK.getStatusCode() == post.getStatus()) {
8.             System.out.println(post.readEntity(TrainBooking.class).getId());
9.         }
10.     }
```

```
1.     private void createTrainBooking(String numTrain, int numberPlaces) {
2.         TrainBooking trainBooking = new TrainBooking();
3.         trainBooking.setTrainId(numTrain);
4.         trainBooking.setNumberPlaces(numberPlaces);
5.
6.         Response post = target.path("bookings").request(MediaType.APPLICATION_JSON_TYPE)
7.                 .post(Entity.entity(trainBooking, MediaType.APPLICATION_JSON));
8.
9.         if (Status.OK.getStatusCode() == post.getStatus()) {
10.             System.out.println(post.readEntity(TrainBooking.class).getId());
11.         }
12.     }
```

- Compléter la méthode callGetTrainBookings permettant de récupérer l'ensemble des réservations de billets de train et de les afficher dans le composant graphique JTextArea.

```
1.     private void callGetTrainBookings() {
2.         // TODO: invoquer le service web REST pour récupérer l'ensemble des réservations
3.         // de billets de train.
4.         // Le résultat doit être transmis dans un objet de type List<TrainBooking>
5.
6.         for (TrainBooking current : result) {
7.             trainBookingsConsole.append(current.getId() + " - " + current.getTrainId() + " -
 " + current.getNumberPlaces());
8.         }
9.     }
```

```
1.     private void callGetTrainBookings() {
2.         List<TrainBooking> result =
 target.path("bookings").request(MediaType.APPLICATION_JSON_TYPE)
3.                 .get(new GenericType<List<TrainBooking>>() {
4.                 });
5.
6.         for (TrainBooking current : result) {
7.             trainBookingsConsole.append(current.getId() + " - " + current.getTrainId() + " -
 " + current.getNumberPlaces());
8.         }
9.     }
```

Même problème pour la classe TrainBooking qui n'existe pas. Copier depuis l'exercice 3, la classe TrainBooking en vous assurant si possible de modifier le nom du package par fr.mickaelbaron.jaxrstutorialexercice4.

```
1. package fr.mickaelbaron.jaxrstutorialexercice5;
2.
3. @XmlRootElement(name = "trainbooking")
4. public class TrainBooking {
```

```
5.
6.      private String id;
7.
8.      @JsonProperty("current_train")
9.      private String trainId;
10.
11.      @JsonProperty("number_places")
12.      private int numberPlaces;
13.
14.      ...
```

Tester votre programme client en invoquant les trois fonctionnalités développées précédemment.

V - Exercice 5 : déployer le service web REST « Interrogation et réservation de billets de train »

V-A - But

- Déployer comme une application Java classique (exécuter depuis un fichier jar).
- Packager un service web REST dans une archive war.
- Déployer un service web REST sur le serveur d'applications Java Tomcat.
- Gérer les problèmes de dépendances.

V-B - Description

Ce cinquième exercice s'intéresse au déploiement du service web REST développé dans les précédents exercices. Nous montrerons le déploiement comme une application Java classique (exécuter depuis un jar) et un déploiement sur le serveur d'application Tomcat (déployer un fichier war). Nous ferons abstraction de l'environnement de développement Eclipse afin d'être le plus proche de l'environnement de production.

Le projet contient tout le code du service web REST.

V-C - Étapes à suivre pour effectuer un déploiement comme une application Java classique

- Saisir la ligne de commande suivante depuis la racine du projet pour compiler et construire le fichier jar du projet.

```
mvn clean package
```

- Saisir la ligne de commande suivante pour démarrer le projet.

```
1. $ java -cp "target/jaxrstutorialexercice5.jar"
 fr.mickaelbaron.jaxrstutorialexercice5.TrainBookingLauncher
2. Exception in thread "main" java.lang.NoClassDefFoundError: javax/ws/rs/core/UriBuilder
3.      at
 fr.mickaelbaron.jaxrstutorialexercice5.TrainBookingLauncher.getBaseURI(TrainBookingLauncher.java:21)
4.      at
 fr.mickaelbaron.jaxrstutorialexercice5.TrainBookingLauncher.<clinit>(TrainBookingLauncher.java:18)
5. Caused by: java.lang.ClassNotFoundException: javax.ws.rs.core.UriBuilder
6.      at java.base/jdk.internal.loader.BuiltinClassLoader.loadClass(BuiltinClassLoader.java:582)
7.      at java.base/jdk.internal.loader.ClassLoaders
$AppClassLoader.loadClass(ClassLoaders.java:178)
8.      at java.base/java.lang.ClassLoader.loadClass(ClassLoader.java:521)
9.      ... 2 more
```

Vous remarquerez que le projet ne démarre pas du fait de l'absence de certaines dépendances. Il est donc obligatoire de fournir les dépendances nécessaires lors de l'exécution (dans le classpath).

- Modifier le fichier *pom.xml* afin d'ajouter le plugin **maven-dependency-plugin** qui permettra de lister toutes les bibliothèques nécessaires.

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>${maven.dependency.version}</version>
    <executions>
        <execution>
            <id>copy-dependencies</id>
            <phase>package</phase>
            <goals>
                 <goal>copy-dependencies</goal>
             </goals>
         </execution>
     </executions>
</plugin>
```

- Saisir les lignes de commande suivantes pour compiler, construire et démarrer le projet.

```
$ mvn clean package
...
$ java -cp "target/jaxrstutorialexercice5.jar:target/dependency/*"
 fr.mickaelbaron.jaxrstutorialexercice5.TrainBookingLauncher
nov. 10, 2018 4:57:35 PM org.glassfish.grizzly.http.server.NetworkListener start
INFO: Started listener bound to [localhost:9993]
nov. 10, 2018 4:57:35 PM org.glassfish.grizzly.http.server.HttpServer start
INFO: [HttpServer] Started.
Jersey app started with WADL available at http://localhost:9993/api/
Hit enter to stop it...
```

V-D - Étapes à suivre pour effectuer un déploiement sur le serveur d'applications Tomcat

Pour un déploiement de service web REST avec Jersey vers un serveur d'applications il est nécessaire de 1) fournir un fichier *web.xml* où il est précisé un objet ResourceConfig à prendre en compte ; 2) construire le fichier war ; 3) fournir au serveur d'application le fichier war.

- Créer une classe TrainBookingApplication de type ResourceConfig.

```
package fr.mickaelbaron.jaxrstutorialexercice5;

public class TrainBookingApplication extends ResourceConfig {

    public TrainBookingApplication() {
        this.registerClasses(TrainBookingResource.class, TrainResource.class);
        this.property(LoggingFeature.LOGGING_FEATURE_LOGGER_LEVEL_SERVER,
 Level.WARNING.getName());
    }
}
```

- Éditer le fichier *src/main/configuration/web.xml* et ajouter la déclaration de la classe TrainBookingApplication.

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    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>
```

```
13.             <param-value>fr.mickaelbaron.jaxrstutorialexercice5.TrainBookingApplication</
param-value>
14.         </init-param>
15.         <load-on-startup>1</load-on-startup>
16.     </servlet>
17.     <servlet-mapping>
18.     <servlet-name>jersey-servlet</servlet-name>
19.         <url-pattern>/api/*</url-pattern>
20.     </servlet-mapping>
21. </web-app>
```

- Éditer le fichier *pom.xml* et à la ligne 102, préciser le chemin et le nom du fichier *web.xml* qui sera utilisé comme descripteur pour l'application web Java.

```
1. <properties>
2.     <project.packaging>war</project.packaging>
3.     <maven.war.webxml>TODO</maven.war.webxml>
4. </properties>
```

```
1. <properties>
2.     <project.packaging>war</project.packaging>
3.     <maven.war.webxml>src\main\configuration\web.xml</maven.war.webxml>
4. </properties>
```

- Saisir la ligne de commande suivante pour compiler et construire le projet vers un fichier war.

```
mvn clean package -P war
```

L'option -P war permet d'utiliser le profil Maven appelé war. Depuis le fichier *pom.xml* examiner la balise `<profiles>`. Cette astuce permet de générer un fichier jar ou un fichier war depuis un même fichier *pom.xml*.

- Saisir la ligne de commande suivante pour télécharger une image Docker de Tomcat.

```
docker pull tomcat:9-jre11-slim
```

- Enfin, saisir la ligne de commande suivante pour créer un conteneur Docker qui permettra de démarrer une instance de Tomcat. Le fichier *jaxrstutorialexercice5.war* contient tous les codes et dépendances de ce projet.

```
1. docker run --rm --name helloworldrestservice-tomcat -v $(pwd)/target/
jaxrstutorialexercice5.war:/usr/local/tomcat/webapps/jaxrstutorialexercice5.war -it -p 8080:8080
 tomcat:9-jre11-slim
2. Using CATALINA_BASE:   /usr/local/tomcat
3. Using CATALINA_HOME:   /usr/local/tomcat
4. Using CATALINA_TMPDIR: /usr/local/tomcat/temp
5. Using JRE_HOME:        /docker-java-home
6. Using CLASSPATH:       /usr/local/tomcat/bin/bootstrap.jar:/usr/local/tomcat/bin/tomcat-
juli.jar
7. ...
```

- Tester le service web REST déployé avec **cURL**.

```
1. $ curl --header "Accept: application/xml" http://localhost:8080/jaxrstutorialexercice5/api/
trains
2. <?xml version="1.0" encoding="UTF-8" standalone="yes"?><trains><train><arrival>Paris</
arrival><departure>Poitiers</departure><departureTime>1250</departureTime><id>TR123</id></
train><train><arrival>Paris</arrival><departure>Poitiers</departure><departureTime>1420</
departureTime><id>AX127</id></train><train><arrival>Paris</arrival><departure>Poitiers</
departure><departureTime>1710</departureTime><id>PT911</id></train></trains>
3.
4. $ curl --header "Accept: application/json" http://localhost:8080/jaxrstutorialexercice5/api/
trains
5. [{"id":"TR123","departure":"Poitiers","arrival":"Paris","departure_time":1250},
{"id":"AX127","departure":"Poitiers","arrival":"Paris","departure_time":1420},
{"id":"PT911","departure":"Poitiers","arrival":"Paris","departure_time":1710}]
```

VI - Conclusion et remerciements

Nous avons vu au travers de quatre exercices comment utiliser l'API JAX-RS. Nous avons notamment insisté sur le développement de services web et de clients utilisés pour appeler les services web de type REST.

Pour aller plus loin, vous pouvez consulter les ressources suivantes :

- **Support de cours SOA** ;
- **Support de cours WSDL** ;
- **Support de cours SOAP** ;
- **Support de cours JAX-WS** ;
- **Support de cours REST** ;
- **Support de cours JAX-RS**
- **Exercices sur le test fonctionnel de services web avec SOAP-UI** ;
- **Exercices sur le développement de services web étendus avec JAX-WS et Netbeans** ;
- **Exercices sur le développement de services web étendus avec JAX-WS, Maven et Eclipse** ;
- **Exercices sur le développement de services web étendus avec JAX-RS et Netbeans**.

Je tiens à remercier **jacques_jean** pour sa relecture orthographique attentive de cet article.