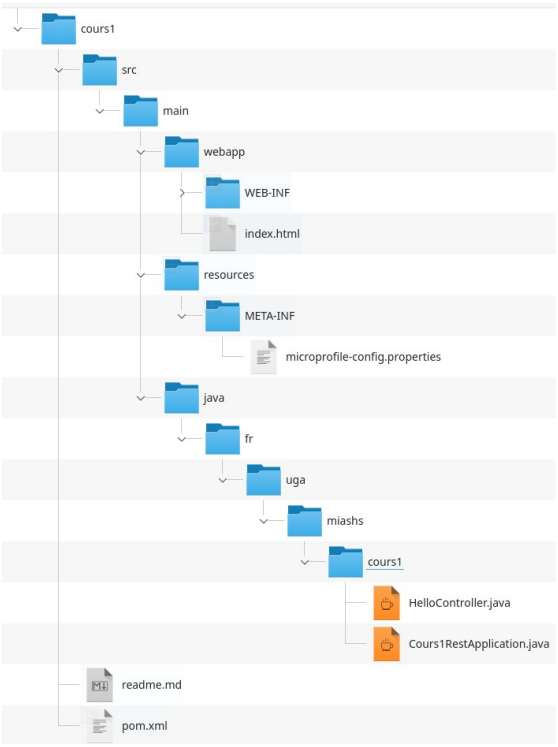
TP1

Découverte de l'environnement de programmation et premiers services

Objectifs : Construire et exécuter un premier service web REST avec l'API JAX-RS et l'environnement d'exécution PayaraMicro. JAX-RS est l'API java pour développer des services Web REST et PayaraMicro est un serveur d'application.

1- Mise en place

Télécharger et décompresser l'application cours1.zip (voir site du cours).
Voici son contenu :



Le répertoire `src/main` contient les sources. Dans `webapp`, vous avez tous les fichiers web statiques (HTML, CSS, librairies Javascript, etc.). Le sous répertoire `WEB-INF` contient généralement des fichiers de configuration et n'est pas adressable de l'extérieur (i.e. non visible depuis le serveur web).
Le dossier `resources/META-INF` contient des fichiers de configuration tels que les paramètres de votre application microprofile ou encore le fichier de configuration du socle de persistance relationnel/objet JPA (`persistence.xml`).
Le dossier `java` contient le code source Java.
Le fichier `pom.xml` est le fichier de configuration de Maven. On y indique entre autre les librairies à utiliser, les différentes manière de packager et exécuter le projet.
Le fichier `readme.md` contient les instructions pour compiler et exécuter le projet.

Exécution de l'application
Suivez pour cela le fichier `readme.md` et vérifiez que cela fonctionne.
Vous devriez voir le message Hello World si vous accédez à l'URL
http://localhost:8080/cours1/data/hello
Ce service web a été implémenté dans la classe `HelloController` de votre projet.
Vous pouvez aller lire le code.

2- Les annotations de base

2.1- `@Path`

L'annotation `@javax.ws.rs.Path` indique une portion de chemin d'URL.
Cette annotation est utilisable sur une classe et sur les méthodes. Si elle est utilisé sur une classe alors toutes les chemins des méthodes de cette classe seront « contenus » dans le chemin de cette classe.

Exemple :
Ajoutez cette classe chemin à votre projet.

```
package fr.uga.miashs.cours1;

import javax.ws.rs.GET;
import javax.ws.rs.Path;

@Path("unCheminDeClasse")
public class Chemins {
   @GET
   @Path("unCheminDeMethode")
   public String uneMethode() {
      return "vous y etes";
   }
}
```

Exécutez votre code et allez sur
http://localhost:8080/cours1/data/unCheminDeClasse/unCheminDeMethode

Ici sur l'URL, `cours1` est le nom de l'application (renseigné dans le `pom.xml`), `data` est le chemin de la partie JAX-RS de l'application déclaré dans la classe `Cours1RestApplication`.

2.2- Les annotations de méthodes HTTP

Dans JAX-RS, il faut spécifier devant les méthodes d'une classe à quel(s) types de méthode HTTP elles correspondent. Les annotations possibles sont les suivantes :

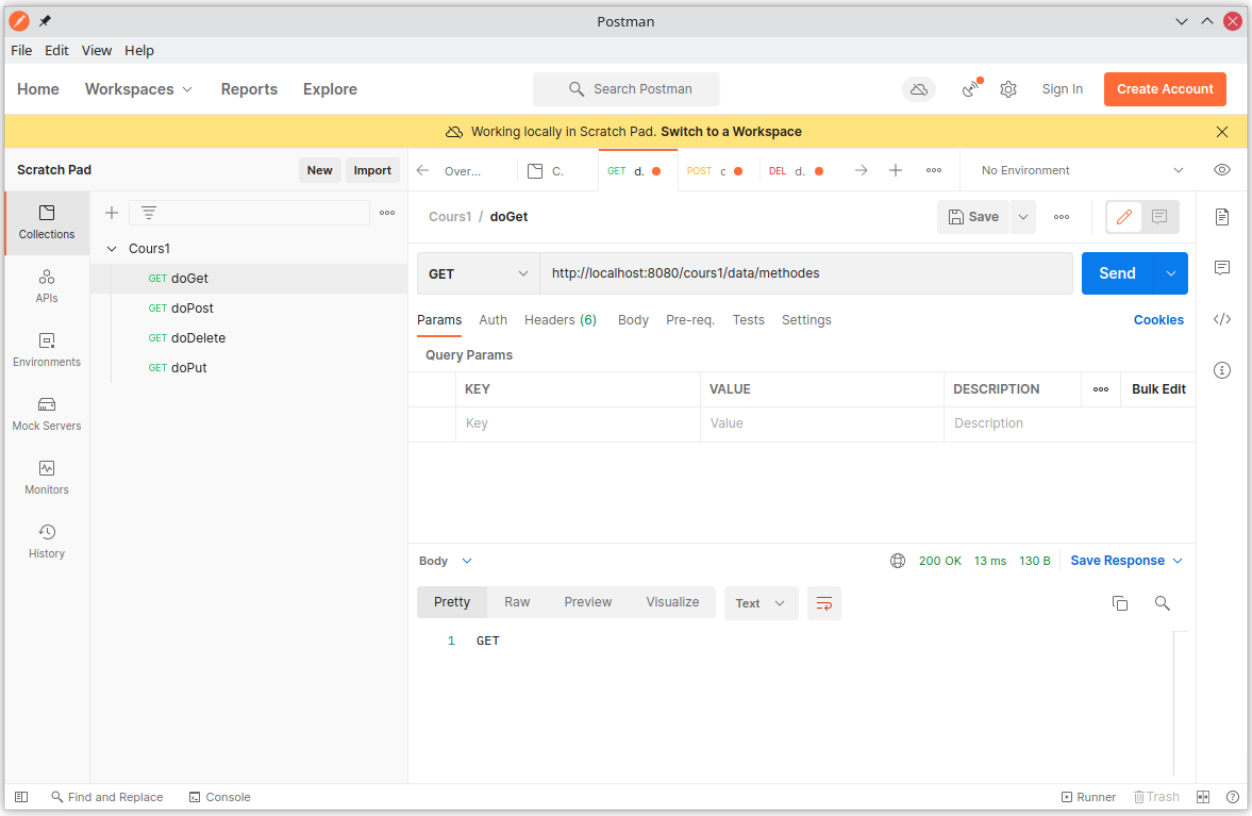
- `@javax.ws.rs.GET`
- `@javax.ws.rs.HEAD`
- `@javax.ws.rs.POST`
- `@javax.ws.rs.PUT`
- `@javax.ws.rs.DELETE`
- `@javax.ws.rs.OPTIONS`

Exemple :
Ajoutez la classe suivante à votre projet :

```
package fr.uga.miashs.cours1;
```

```java
import javax.ws.rs.*;

@Path("methodes")
public class Methodes {
   @GET
   public String doGet() { return "GET";  }

   @POST
   public String doPost() { return "POST";  }

   @DELETE
   public String doDelete() { return "DELETE";  }

   @PUT
   public String doPut() { return "PUT";  }
}
```

Exécutez votre code, et ouvrez l'application Postman.
Dans Postman, créez une nouvelle collection « Cours1 » et créez 4 requêtes correspondantes aux différentes méthodes et à l'URL http://localhost:8080/cours1/data/methodes :



Essayez également la même URL avec une méthode non implémentée dans votre code, comme par exemple PATCH. Que se passe-t-il ? A quoi correspond le résultat ? Quel est le code HTTP du résultat ? Essayez maintenant avec la méthode OPTION. Qu'obtenez vous ?

Les paramètres de chemin
Lorsque l'on développe des services REST, chaque ressource est identifiée par une URI.
Par exemple, on peut imaginer qu'un utilisateur de notre application soit désigné par l'URI :
http://localhost:8080/cours1/data/user/1

Par convention, généralement cela signifie : « l'objet de type user ayant l'identifiant 1 ».
Le « 1 » ici est un paramètre que l'on doit pouvoir extraire. On appelle cela un paramètre de chemin.
Les paramètres de chemin sont déclarés dans les Path avec un nom de variable encadré par des accolades et injecté dans un paramètre de méthode grâce à l'annotation @PathParam.
Exemple :

```java
package fr.uga.miashs.cours1;

import javax.ws.rs.*;

@Path("/users/{id}")
public class UserController {
   @GET
   public String get(@PathParam("id") long id) {
     return "Lit l'utilisateur avec l'id "+id;
   }

   @PUT
   public String createOrUpdate(@PathParam("id") long id) {
      return "Mise à jour de l'utilisateur avec l'id "+id;
   }

   @DELETE
   public String delete(@PathParam("id") long id) {
      return "Effacer l'utilisateur avec l'id "+id;
   }

   @GET
   @Path("amis")
   public String lesAmis(@PathParam("id") long id) {
      return "retourne les amis de l'utilisateur avec l'id "+id;
   }
}
```

Exécutez votre code et testez les différentes méthodes via Postman.
Vous remarquerez que la dernière méthode lesAmis est accesssible par exemple via `.../users/2/amis`
Testez maintenant avec une URL qui ne contient pas un entier comme identifiant, comme par exemple : http://localhost:8080/cours1/data/user/aaa
Testez également : http://localhost:8080/cours1/data/user/-888

Vous verrez que le contrôle de type permet de faire une partie du travail de validation des URL mais qu'on peut être parfois amené à restreinte plus le type. Par exemple, si jeux forcer l'id d'un utilisateur à être positif, il faudra définir le chemin comme suit : `@Path("/users/{id : [0-9]+}")`
Si l'on désire un code postal sur 5 chiffres, on utilisera par exemple `{cp : [0-9]{5}}`.

2.3- @Consumes / @Produces

Les exemples de services vus jusqu'à présent ne consommaient aucune données et produisaient une simple chaîne de caractères.

Généralement, les services web consomment et produisent des données aux formats JSON ou XML (mais peuvent aussi retourner des fichiers image ou autre…).

Dans les requêtes et réponses HTTP, le type de contenu doit normalement être précisé avec une entête (header) HTTP Content-type qui donne le type MIME du contenu envoyé (voir https://fr.wikipedia.org/wiki/Type_de_m%C3%A9dias).

Avec JAX-RS, le type de contenu peut être précisé via les annotations `@Consumes` et `@Produces`.

Voici un exemple basé sur le code précédent :

```
package fr.uga.miashs.cours1;

import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;

@Path("/users/{id}")
public class UserController {
   @GET
   @Produces(MediaType.TEXT_HTML)
   public String get(@PathParam("id") long id) {
      return "<html><body>Lit l'utilisateur avec l'id "+id+"</body></html>";
   }
}
```

Si vous enlevez l'annotation `@Produces(MediaType.TEXT_HTML)` que Postman verra la réponse comme du texte et non du HTML (voir la coloration syntaxique et le type de contenu dans Postman).

2.4- Data Binding

Pour aller plus loin, on peut même voir qu'avec les annotations `@Produces` et `@Consumes`, nous pouvons paramétrer dans quels formats seront transformées et envoyées des instances de classes Java.

Exemple : Envoyer et recevoir des instances d'une classe `Person` en JSON

1- Créer une classe `Person` avec quelques attributs (`id`, `mail`, `nickname`), des getters et setters pour chaque attributs, et un constructeur prenant en paramètre ces différents attributs.

2 – Copiez et adaptez éventuellement le code ci-dessous :

```
package fr.uga.miashs.cours1;

import javax.inject.Singleton;
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import java.net.URI;
import java.util.*;
import java.util.concurrent.atomic.AtomicLong;
```

```java
@Path("/persons")
// @Singleton est utilisée pour garantir q'une seule instance de cette classe sera isntanciée.
// sinon une instance est crée par appel et on perd le contenu de la Map
@Singleton
public class PersonController {

    private Map<Long,Person> data;
    private AtomicLong idGen;

    public PersonController() {
        idGen = new AtomicLong(0);
        data = new HashMap<>();
        long id = idGen.getAndIncrement();
        data.put(id,new Person(id, "toto@dudule.org", "toto"));
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("/{id: [0-9]+}")
    public Person get(@PathParam("id") long id) {
        if (data.containsKey(id)) {
            return data.get(id);
        }
        // Si id inconnu alors on envoie une 404
        throw new NotFoundException();
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response add(Person p,  @Context UriInfo uriInfo) {
        long id = idGen.getAndIncrement();
        p.setId(id);
        data.put(id,p);
        // on construit l'URI correspondant à la personne
        URI location = uriInfo.getRequestUriBuilder()
                .path(String.valueOf(p.getId()))
                .build();
        // On retourne la réponse
        return Response.created(location).entity(p).build();
    }
}
```

3 – Testez !

Un GET sur http://localhost:8080/cours1/data/persons/0

Un POST sur http://localhost:8080/cours1/data/persons avec comme contenu (body dans Postman) :

```
{
        "email": "tutu@dudule.org",
        "nickname": "tutu"
}
```

Notez qu'on aurait éventuellement pu retourner seulement une instance de `Person` dans la méthode add. Ici on a créé un objet « `javax.rs.code.Response` ». Ce type d'objet permet de configurer plus finement la réponse HTTP qui sera envoyé. On peut par exemple ajouter un code de statut différent , ou encore ajouter des entêtes.

Dans l'exemple ci dessus, l'appel à `created(…)` envoie un code de réponse HTTP 201 au lieu de 200 (voir le status dans Postman),et ajoute une entête (header) « location » avec l'URI de la nouvelle ressource crée.