Microservices Oriented Software Architectures

States

Today, we are going to see how to set up a RESTful webservice, with the Jersey library. We will use JSON as the data transport medium.

Goals

Master the notion of RESTful web service on the server side and on the client side.

Creation of a Helloworld webservice (server)

Step n°1: Creation of the Jersey project via maven

Like other TPs, create a jersey project with maven. Here is the archetype to use:

```
com.sun.jersey.archetypes:jersey-quickstart-webapp
```

with the following options:

```
groupId: alom
artifactId: jersey-helloworld-server
version: 1.0-SNAPSHOT
package: alom.server
```

Import the project into Eclipse.

Step n°2: Discovery of the generated class

Open the generated class: `MyRessource`

This class is a resource in the Jersey sense of the term, because it is annotated by:
`@Path("/myresource")`

`The @Path` annotation indicates that this class is a Jersey resource, accessible at `/myresource` from the URL to access the Jersey instance. We will see this concept in the next step.

In this "resource" class, there is a `getIt()` method , which returns the string `"Hi there!"` . This method is doubly annotated by `@GET` and `@Produces("text/plain")`

`The @GET` annotation means that the method will be called following an HTTP GET on the resource. It is possible to have several `@GET` annotations on different methods, provided that the methods are themselves annotated with `@Path`, to distinguish the different resources on which the HTTP GETs apply.

Example #1: Correct

```
@Path("example")
public class JerseyExample { @GET @Path("/hi")    public String hi() {        return "Hi";    } @GET
@Path("/hello")    public String hello() {        return "Hello";    } }
```

An HTTP GET to the url http://xxx/xxx/example will not work.
An HTTP GET at url http://xxx/xxx/example/hi will call `hi()` .
An HTTP GET at url http://xxx/xxx/example/hello will call `hello()` .

Example 2: incorrect

```
@Path("example")
public class JerseyExample { @GET    public String hi() {        return "Hi";    } @GET    public String
hello() {        return "Hello";    } }
```

An HTTP GET on the url http://xxx/xxx/example will not work because Jersey does not know which of the 2 methods to call.

Example n°3: possible/correct

```
@Path("example")
public class JerseyExample { @GET    public String hi() {        return "Hi";    } @GET @Path("/hello")
   public String hello() {        return "Hello";    } }
```

An HTTP GET at url http://xxx/xxx/example will call `hi()` .
An HTTP GET to the url http://xxx/xxx/example/hi will not work.
An HTTP GET at url http://xxx/xxx/example/hello will call `hello()` .

`The @Produces` annotation indicates the HTTP ContentType produced by the call to this method. Here the `"text/plain"` indicates that the String will be returned as plain text.

Step n°3: Implementation of the JSON mode in Jersey and modification of the MyRessource class

Open in eclipse the file located in `jersey-helloworld-server /src/main/webapp/WEB-INF/web.xml` , you can notice:

- A Jersey Web Application servlet: This is the Jersey servlet, which allows you to run Jersey in your tomcat
- A mapping of this servlet on the URL `/webresources/*` : which indicates that your url will be of the form http://xxx/xxx/webresources/*

The Jersey servlet uses an initialization parameter: `com.sun.jersey.config.property.packages` : which indicates the Java packages in which Jersey should look for resource classes (annotated by `@Path`)

Add the following initialization parameter to the Jersey servlet:

```
        <init-param>
            <param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>
            <param-value>true</param-value>
        </init-param>
```

This parameter enables the functionality to serialize/deserialize Java objects to JSON. We will now implement it.

Create a java `Hello` class , which contains a private attribute, of the `String` class , named `string` . Create a constructor that takes the value of the `string` attribute as a parameter, then the getter and the setter corresponding to this attribute.

`Then in the MyRessource` resource class , replace the `getIt()` method and its annotations with the following code:

```
@GET
@Path("/hello")
@Produces("application/json")
public Hello hello() {
     return new Hello("Hello");
}
```

Step 4: Launching the project in Tomcat and first tests

Add the project to the Tomcat instance (Add and Remove... menu by right-clicking on the Tomcat instance).

Then access your resource by going to the URL: http://localhost:8080/jersey-helloworld-server/webresources/myresource/hello

You should get this:
`{"string":"Hello"}`

Creation of a Helloworld webservice (client)

Step n°1: Creation of the project

Like the server, create another jersey project with maven. Here are the recommended options:

```
groupId: alom
artifactId: jersey-helloworld-client
version: 1.0-SNAPSHOT
package: alom.client
```

Import the project into Eclipse. Remove the `MyResource` class which is a useless class on the client side (it is only useful for the server). In the `pom.xml` file, you must also remove the <scope>test</scope> for the `jersey-client dependency .`

Step 2: Creation of RESTful WebServices classes

One of the major advantages of RESTful WebServices is also its main drawback: flexibility. Indeed, unlike XML WebServices, it is possible to easily add methods or parameters to a RESTful WebService, without the client being impacted. On the other hand, RESTful services do not natively benefit from interface description standards (such as WSDL) and therefore from the resulting automatic code generation tools.

It is therefore necessary to recreate (or copy) the `Hello` class on the client side (you can name it as you wish, there is no constraint on the name), respecting the following constraints (from RESTful mechanisms):

- an "empty" constructor (i.e. without parameter, for instantiation by WebServices libraries)
- a getter and setter: `getString()` and `setString(String string)`
- the class must be annotated with `@XmlRootElement` in order to indicate to the framework that the class is serializable

Step n°3: Creation of the client itself

Create a client class `HelloWorldClient` composed of:

- `A service` attribute of the `WebResource` class (`com.sun.jersey.api.client` package)
- A builder :
  - creating a Jersey client with the default configuration: `Client client = Client.create(new DefaultClientConfig());`
  - assigning to the service attribute, the resource created from the client taking the URL of the resource as a parameter: `this.service = client.resource("http://localhost:8080/jersey-helloworld-server/webresources/myresource ");`
- The `main` method to launch a Java application

Now, create a `callHello()` method , which will call the `hello()` method on the server side, using the following elements:

- `The service` attribute
- `the path` method , allowing to indicate the additional path to add to the URL where the service is located to call a given method
- `the accept` method , allowing you to indicate that you expect the `MediaType.APPLICATION_JSON_TYPE` type in return from your call (allowing you to obtain JSON in response to the call)

- `the get` method , allowing to call the resource using the HTTP GET. This method takes as a parameter the type of return on which to "map" the information coming from the server

Now, in your `main` method , instantiate your `HelloWorldClient` class , call the `callHello()` method and display the result.

Step 4: Using the POST HTTP method

On the Server side, create a `sayHello` method , taking a `Hello` type object as a parameter . This method returns a `Hello` type object , whose `string` attribute equals `"Hello "` + the value of the `string attribute of the Hello` object passed as a parameter.
At the annotation level, you can use the same as for the `hello` method , replacing `@GET` by `@POST` , and adding `@Consumes(MediaType.APPLICATION_JSON)` which indicates that the method parameters must be provided in JSON format when of the call.

On the Client side, create the method that calls the `sayHello` method , using the same series of method calls made on the service attribute, except for the `get` method , and adding the following calls:

- `the type` method , allowing to indicate that you are sending the data with the `MediaType.APPLICATION_JSON_TYPE` type (allowing to send the parameters in JSON format)
- `the post` method , allowing to call the resource using the HTTP POST. This method takes as a parameter the type of return on which to "map" the information coming from the server, as well as the object to pass as a parameter of your method

Now, in your `main` method , call the `sayHello` method and display the result. If you have errors, try to understand them and correct them in order to obtain the desired result.

Continuation and end **Here begins The real Project**

For this last lab, create a mini application made up of resources: student, group, subject, mark. In particular, you will need to use the `@Path` and `@PathParam` annotations . On this application, we must in particular be able to:

- Perform "CRUD" operations (Create/Retrieve/Update/Delete => Create/Retrieve/Update/Delete => @POST/@GET/@PUT/@DELETE) on:
  - student
  - group
  - subject
- Add & remove students from groups (a student can only be part of one group)
- Create & delete a link between group and subject (a group follows the lessons of several subjects, you can take inspiration from the portal to create your datasets)
- Give, modify & delete grades in a subject to a student
- Display the table of results (all student grades, grouped by subject and group

For this lab, you must implement the exceptions internal to your application in the same way as in the previous lab. However, you can try/catch all of the exceptions in your front classes (the resources exposed by Jersey) and only display a stack trace on the server side and `return null;` when a result is expected. (We don't throw an exception through the API)

Modified on: Tuesday 28 September 2021, 00:33

◄ Render

Go to...

Render ►