Development of Distributed Applications
Seminar 4 – Service-oriented middleware: Building a JSON REST API

November 2015

Davinia Hernández-Leo, David Llanos Santos

Description

The aim of this seminar is to develop an example of REST API that consumes messages and produces responses in JSON format. This application is programmed using an open source framework for developing RESTful Web Services in Java.

The operation of the application is as follows:

- The server module will publish an API with 3 services (add/get/list all).
- Clients will make requests to the API to validate its functionality.

Objectives

- Build a small REST API using Jersey as framework for developing RESTful Web Services in Java.
- Build a small REST client to test the API.
- Use Maven to manage dependencies (Java libraries).

Before the seminar session

Before the seminar session it is necessary to read the following sections of "Project (and seminar 4) technologies" theme: Maven, REST, Jersey and JSON.

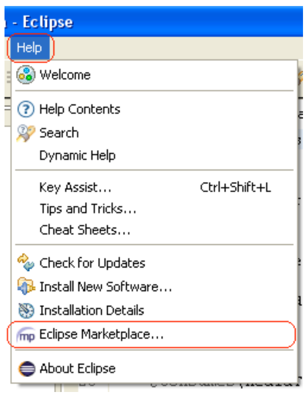
Development

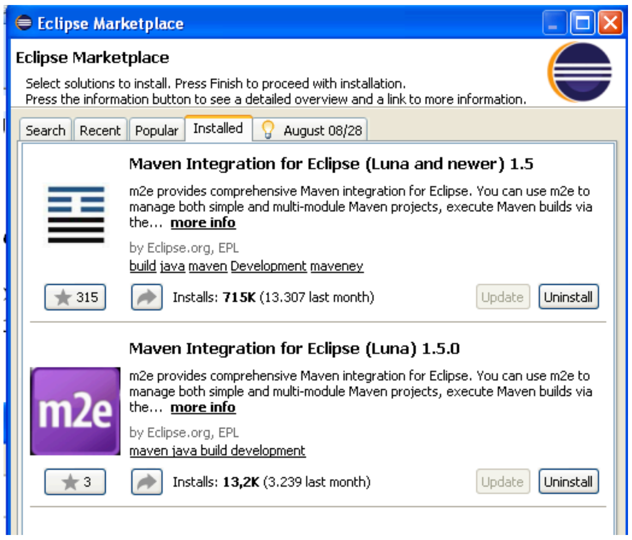
Next sections should be followed to build the application. The manual is divided into three parts that develop the functionality of the REST API incrementally:

- Managing Java dependencies with MAVEN.
- Building a REST API with Jersey framework and testing it using a web browser.
- Connecting a REST client to the API to test all the published services.

Managing Java dependencies with MAVEN

As mentioned in the introduction, we use maven to manage application dependencies. First, download the maven plugin for eclipse using the Eclipse Marketplace (you can find it in help menu). Search for Maven Integration for Eclipse plugin and install it.





Eclipse Marketplace

Select solutions to install. Press Finish to proceed with installation.
Press the information button to see a detailed overview and a link to more information.

Maven Integration for Eclipse (Luna and newer) 1.5

m2e provides comprehensive Maven integration for Eclipse. You can use m2e to manage both simple and multi-module Maven projects, execute Maven builds via the... **more info**

by Eclipse.org, EPL

build java maven Development maveney

315 Installs: **715K** (13.307 last month) Update Uninstall

Maven Integration for Eclipse (Luna) 1.5.0

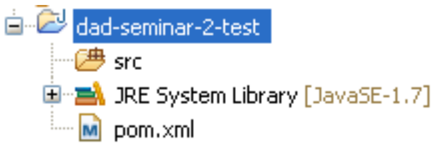
m2e provides comprehensive Maven integration for Eclipse. You can use m2e to manage both simple and multi-module Maven projects, execute Maven builds via the... **more info**

by Eclipse.org, EPL

maven java build development

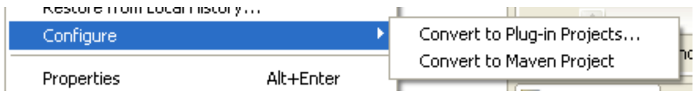
3 Installs: **13,2K** (3.239 last month) Update Uninstall

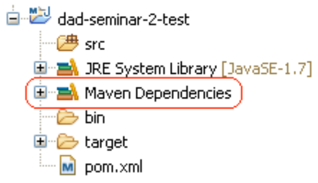
After installation, create a new Java project, and add a file called pom.xml to the root of the project. The content of the file could be as follows:



```xml
<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/xsd/maven-4.0.0.xsd">
        <modelVersion>4.0.0</modelVersion>
        <groupId>upf.dad.s2</groupId>
        <artifactId>JAXRS-Seminar2</artifactId>
        <version>0.0.1-SNAPSHOT</version>

        <dependencies>
                <!-- jersey -->
                <dependency>
                        <groupId>org.glassfish.jersey.containers</groupId>
                        <artifactId>jersey-container-servlet</artifactId>
                        <version>2.12</version>
                </dependency>
                <!-- JDK Http Container for jersey -->
                <dependency>
                        <groupId>org.glassfish.jersey.containers</groupId>
                        <artifactId>jersey-container-jdk-http</artifactId>
                        <version>2.12</version>
                </dependency>
                <!-- Use moxy as JSON provider -->
                <dependency>
                        <groupId>org.glassfish.jersey.media</groupId>
                        <artifactId>jersey-media-moxy</artifactId>
                        <version>2.12</version>
                </dependency>
                <!-- JAX-RS Client -->
                <dependency>
                        <groupId>org.glassfish.jersey.core</groupId>
                        <artifactId>jersey-client</artifactId>
                        <version>2.12</version>
                </dependency>
        </dependencies>
</project>
```

After adding the project object model for Maven, it is necessary to convert the Java Project to Maven Project: Click the right mouse button on the project name and access the **Configure** option to convert it. After doing it, check that Maven Dependencies has been added to the Libraries section of the project.





Building REST API with Jersey framework and testing it using a web browser

First of all, create the object (resource) we will use in the interface to operate with it (add / get / list all). This is the resource that clients will manipulate using the REST API. Remember to add constructors (empty and with arguments), getters, setters and toString methods to the class.

```java
package upf.dad.s3.data;

public class Item {

        private String name;
        private int price;
        private int quantity;

        Add constructors (empty and with arguments), getters, setters and toString
}
```

After creating the object, create the Services class which contains the methods we will publish in the API. The clients will use URLs and HTTP operations GET, PUT, POST and DELETE to manipulate the resource (Item) we have published in the interface.

You can use the following code to publish two methods: create item and get item by name

- All the methods of the class will be published with the URL prefix /item: `@Path("/item")`
- Each method will make available its functionality defining the HTTP operation to access it (`@POST`, `@GET`) and also the URL suffix (`@Path("/add")`, `@Path("/get/{name}")`).
- The annotations `@Consumes(MediaType.APPLICATION_JSON)` and `@Produces(MediaType.APPLICATION_JSON)` define the media types that the method of a resource can accept or produce.

You can extend this information at the following URLs:

- JAX RS javadoc annotations used to create RESTful service resources: http://docs.oracle.com/javaee/6/api/javax/ws/rs/package-summary.html
- Chapter 3 of Jersey documentation: https://jersey.java.net/documentation/latest/jaxrs-resources.html

```
package upf.dad.s3.server;

@Path("/item")
public class Services {

        static List<Item> items = new ArrayList<Item>();

        @POST
        @Path("/add")
        @Consumes(MediaType.APPLICATION_JSON)
        @Produces(MediaType.APPLICATION_JSON)
        public Response create(Item item) {
                items.add(item);
                return Response.status(200).entity(item).build();
        }

        @GET
        @Path("/get/{name}")
        @Produces(MediaType.APPLICATION_JSON)
        public Item get(@PathParam("name") String name) {
                for (Item item : items) {
                        if (item.getName().equalsIgnoreCase(name)) {
                                return item;
                        }
                }
                return null;
        }

        // Add method get all items

}
```

Add a new method to the previous class to return all items.
Modify the response of the get method to return a 404 (Not found) instead of null when the item is not found in the list.

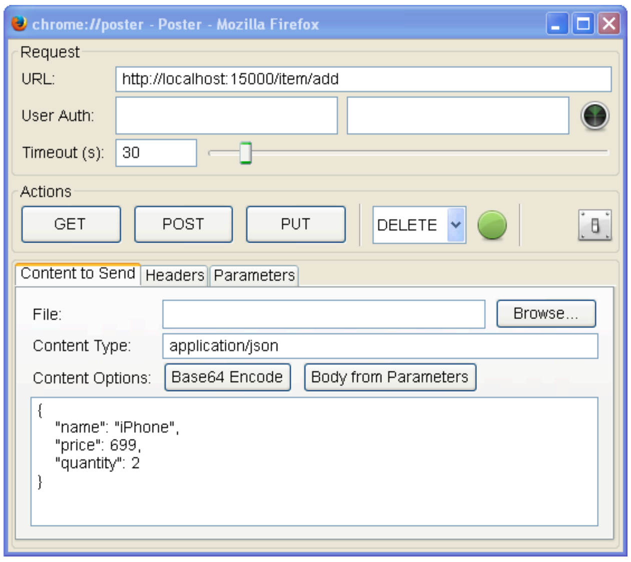
Finally, we only need create and start a JDK HttpServer with the Jersey application configured in the ResourceConfig and deployed in the given URI:

```
package upf.dad.s3.server;

public class RestServer {

        public static void main(String[] args) throws IOException {
                URI baseUri = UriBuilder.fromUri("http://localhost/").port(15000).build();
                ResourceConfig config = new ResourceConfig(Services.class);
                HttpServer server = JdkHttpServerFactory.createHttpServer(baseUri, config);
                System.out.println("Server started...");
        }

}
```

Launch the server and add a new item sending a POST request with content type `application/json` and a JSON object like this:

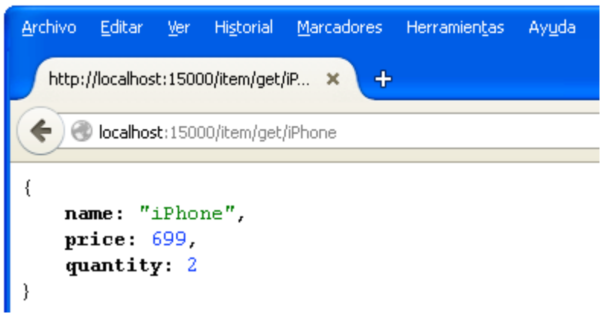
```
        {
            "name": "iPhone",
            "price": 699,
            "quantity": 2
        }
```

You can use Poster Firefox plugin to make the POST request (https://addons.mozilla.org/es/firefox/addon/poster/) and the web browser to get the item you have just added.



URL: http://localhost:15000/item/add
User Auth:
Timeout (s): 30
File:
Content Type: application/json
Content Options:

```
{
  "name": "iPhone",
  "price": 699,
  "quantity": 2
}
```



localhost:15000/item/get/iPhone

```
{
    name: "iPhone",
    price: 699,
    quantity: 2
}
```

Connecting a REST client to the API to test all the published services

Use the following example of REST client using Jersey framework to add two new items to the server and get one of them.

```java
package upf.dad.s3.client;

public class RestClient {

        public static void main(String[] args) {

                // Items
                Item item1 = new Item("iPhone6", 699, 10);
                Item item2 = new Item("iPhone5", 399, 500);

                Client client = ClientBuilder.newClient();
                WebTarget targetAdd = client.target("http://localhost:15000").path("item/add");

                // Add items
                Item response1 = targetAdd.request(MediaType.APPLICATION_JSON_TYPE)
                                .post(Entity.entity(item1, MediaType.APPLICATION_JSON), Item.class);
                Item response2 = targetAdd.request(MediaType.APPLICATION_JSON_TYPE)
                                .post(Entity.entity(item2, MediaType.APPLICATION_JSON), Item.class);

                WebTarget targetGet = client.target("http://localhost:15000").path("item/get/iPhone6");

                // Get item by name
                Item item = targetGet.request(
                                MediaType.APPLICATION_JSON_TYPE).get(new GenericType<Item>() {});
                System.out.println("Item: " + item);

        }
}
```

Add a new request in the client to retrieve all the items calling the get all items method you created in the previous section.

Seminar assessment

For the assessment of the seminar, it is necessary to show the operation of the application to the professor:

- Launch the server and then the client.
- The client must create two new items in the server and make a final request to get all the items in the server using the new method (get all items).
- The server must return a 404 Http Response in case the item does not exist.

The operation can be shown along the duration of Seminar 4 or at the beginning of the first session of the Project (sesión de prácticas).