RESTful APIs with Jersey

Short introduction to REST web services development with Maven + Jersey

REST (Representational State Transfer)

- Software architectural style for web services
- Alternative to heavier web service representation (e.g. SOAP)
- Ease and simplify communication between computers
- Fast performance
- Reliability
- Stateless

Architectural properties

- **Client-server architecture.** Separates client interface from the data and operations. Improves scalability.
- **Stateless.** No context is maintained by the server between different requests. Each requests contains all the required information.
- **Uniform interface.** Share a common mechanism for access (HTTP) that make them easier to use.
- **RESTful services common aspects:**
  - Base URI: `http://someuri.com/api/`
  - Standard HTTP methods: `GET, POST, PUT, PATCH, DELETE`
  - Media types for transferred data: `application/text`, `application/json`, etc.

HTTP Methods

	URI	
HTTP METHODS	**Collection resource, such as `https://api.example.com/collection/`**	**Member resource, such as `https://api.example.com/collection/item3`**
GET	*Retrieve* the URIs of the member resources of the collection resource in the response body.	*Retrieve* representation of the member resource in the response body.
POST	*Create* a member resource in the collection resource using the instructions in the request body. The URI of the created member resource is *automatically assigned* and returned in the response *Location* header field.	*Create* a member resource in the member resource using the instructions in the request body. The URI of the created member resource is *automatically assigned* and returned in the response *Location* header field.
PUT	*Replace* all the representations of the member resources of the collection resource with the representation in the request body, or *create* the collection resource if it does not exist.	*Replace* all the representations of the member resource or *create* the member resource if it does not exist, with the representation in the request body.
PATCH	*Update* all the representations of the member resources of the collection resource using the instructions in the request body, or *may create* the collection resource if it does not exist.	*Update* all the representations of the member resource, or *may create* the member resource if it does not exist, using the instructions in the request body.
DELETE	*Delete* all the representations of the member resources of the collection resource.	*Delete* all the representations of the member resource.

Java API for RESTful Web Services (JAX-RS)

- JAX-RS is a Java specification for programming RESTful APIs.
- Introduced in Java 5.0
- Uses Java annotations (@GET, @PATH, etc)
- There could exist different implementations of this specification.
- Project Jersey (https://github.com/eclipse-ee4j/jersey/) is the reference implementation for this specification.

Creating the project skeleton with Maven

- Using a maven archetype:

```
mvn archetype:generate -DarchetypeArtifactId=jersey-quickstart-grizzly2 \
-DarchetypeGroupId=org.glassfish.jersey.archetypes -DinteractiveMode=false -DgroupId=com.example \
-DartifactId=simple-service -Dpackage=com.example -DarchetypeVersion=2.30.1
```

- This project uses grizzly as the embedded HTTP server for testing.
- The archetype creates a sample web service and some sample unit tests.
  - Compile and run tests: `mvn test`
  - Run application on test server: `mvn exec:java`
  - Create war package for deployment on external servlet container: `mvn clean package`
- Detailed documentation about the process and the contents of the newly created project:

  https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/getting-started.html

Adding a new method to obtain users (JSON)

- Create a POJO to represent users:

```
package com.example.pojo;


public class User {

		private int code;
		private String name;
		private String surname;

		//Default public constructor required for serialization
		public User() {

		}

		public User(int code, String name, String surname) {
			this.code = code;
			this.name = name;
			this.surname = surname;
		}

		public void setCode(int code) { this.code = code; }
		public int getCode() { return code;  }
		public void setName(String name) { this.name = name;	}
		public String getName() {  return name;		}

		public String getSurname() {return surname; }
		public void setSurname(String surname) { this.surname = surname; }
}
```

Adding a new method to obtain users (JSON)

- Create the a new servlet resource (Users.java)
- Add support for json serialization in pom.xml
- Compile and launch the embedded server:
  - mvn compile exec:java

Access to http://localhost:8080/myapp/users

```
package com.example;

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

import java.util.List;
import java.util.ArrayList;

import com.example.pojo.User;

@Path("users")
public class Users {

	@GET
	@Produces(MediaType.APPLICATION_JSON)
	public List<User> getUsers() {
		// This data could be retrieved from a database
		List<User> users = new ArrayList<User>();
		users.add(new User(0, "John", "Smith"));
		users.add(new User(1, "Isaac", "Newton"));
		users.add(new User(0, "Albert", "Einstein"));

		return users;
	}
}
```

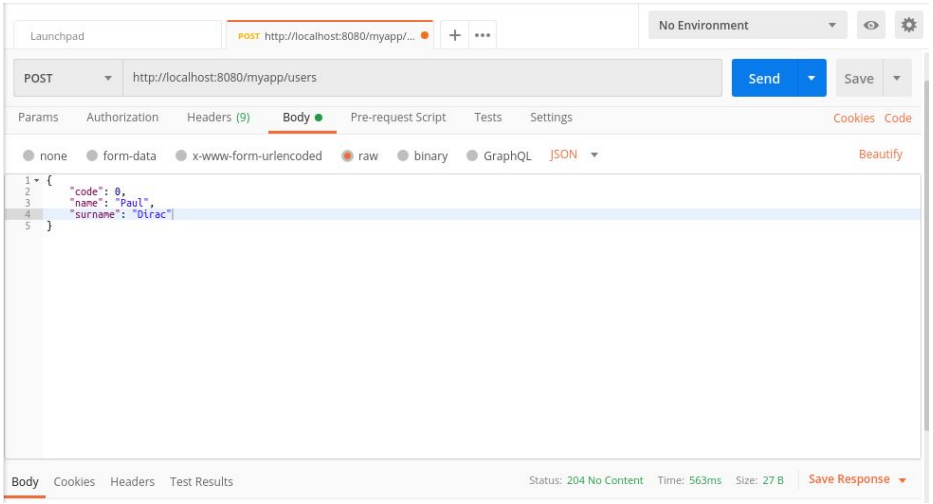
Add a method to add new user

- Add the code to users resource
- Add support for json serialization in pom.xml
- Compile and launch the embedded server:
  - mvn compile exec:java

```
...

@Path("users")
public class Users {

		@GET
		@Produces(MediaType.APPLICATION_JSON)
		public List<User> getUsers() {
			// This data could be retrieved from a database
			List<User> users = new ArrayList<User>();
			users.add(new User(0, "John", "Smith"));
			users.add(new User(1, "Isaac", "Newton"));
			users.add(new User(0, "Albert", "Einstein"));

			return users;
		}

		@POST
		@Consumes(MediaType.APPLICATION_JSON)
		public void addUser(User user) {
		 System.out.println("Adding a new user: " + user.getName() +  " " + user.getSurname());
		  //Save the data to a database
		}
}
```

Easy testing of REST methods with Postman

- Install postman from https://www.postman.com/



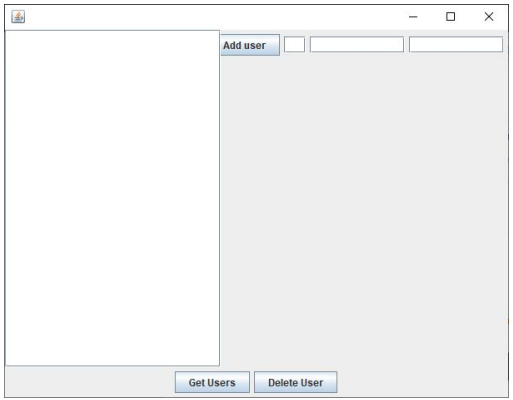
Add a new method to delete a user

- Añadimos un método DELETE para borrar usuarios existentes.
- En este ejemplo, simulamos el borrado:
  - Si se borra el usuario 10 el resultado es correcto
  - Si se borra otro código de usuario se debe notificar al cliente el error con una respuesta específica.
- En este caso no vamos a recibir un objeto JSON, sino una variable en en la URL de la petición.

```
@DELETE
@Path("/{code}")
public Response deleteUser(@PathParam("code") int code) {
	if (code == 10) {
		System.out.println("Deleting user...");
		return Response.status(Response.Status.OK).build();
	} else {
		return Response.status(Response.Status.NOT_FOUND).build();
	}
}
```

Client application

- Copy the file `ClientApp.java` into the `es.deusto.sqp.client` package
- Modify the pom.xml to launch the client application with an specific profile named `client`
- Test that client application is launched with the command:
  - `mvn compile exec:java -Pclient`




```
<profiles>
  <profile>
    <id>client</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <configuration>
            <mainClass>es.deusto.spq.client.ClientApp</mainClass>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

Creation of Jersey Client object

- Create a javax.ws.rs.client.Client object using a builder.
- We must specify the URL where the server application is located.
- New web target routes can be created adding the path to the main URL (p.e. “users”)

```
client = ClientBuilder.newClient();

final WebTarget appTarget = client.target("http://localhost:8080/myapp");
final WebTarget usersTarget = appTarget.path("users");
```

GET request from client

- When the users presses the “Get users” button we want to obtain the users from the server and show them in the JList component of the UI application.
- The client does a GET request to the server method that obtains the user list in JSON format.
- We must specify the format of the data that the client is expecting from the server.
- The client processes the JSON response and transforms it to a List<User> adding it to the visual JList component.

```
GenericType<List<User>> genericType = new GenericType<List<User>>(){};
 List<User> users = usersTarget.request(MediaType.APPLICATION_JSON).get(genericType);

userListModel.clear();
for (User user : users) {
      userListModel.addElement(user);
}
```

Sending data to the server with a POST

- We want to send a new user to the server when the user presses the “Add User”.
- First, we build a new User object containing the data to be sent to the server.
- A POST request is performed using the builtin methods.
- There are two different formats that must be specified:
  - The Entity media type specifies how the data is serialized to the server.
  - The request media type specifies how the server’s response (if any) is going to be sent to the client after processing the POST.

```
User newUser = new User(Integer.parseInt(codeTextField.getText()), nameTextField.getText(), surnameTextField.getText());
usersTarget.request(MediaType.APPLICATION_JSON).post(Entity.entity(newUser, MediaType.APPLICATION_JSON));
```

Error management

- When the client and the server communicate with each other there could arise some errors.
- The server could receive a request that cannot be correctly process (due to some internal errors or because some referenced object could not be found).
- The server should notify the client if any problem was produced when processing the request.
- This notification is usually made using the standard HTTP status codes. However, it could be possible to serialize specific application error messages.
- The client will process the server response performing the required actions (e.g. showing an error window).
- Another common error is when the client cannot connect with the server (the server is down or there is some communication problem).

Problems with the server connection

- The client should detect whenever there is a problem communicating with the server.
- When a communication problem with the server occurs the Jersey client library produces an exception of type `javax.ws.rs.ProcessingException`.
- This exceptions can be catched to perform some action.

```
try {
 . . .
} catch (ProcessingException ex) {
        JOptionPane.showMessageDialog(ClientApp.this, "Error connecting with server", "Error message", ERROR_MESSAGE);
}
```