



UNIVERSITÀ DEGLI STUDI DI GENOVA

REST Web Services: JSR, REST Web Service and Client Development with Jersey, JSON and GSON

Giulia Martino, Fabio Valla

January 2020

Contents

1	Introduction	2
1.1	REST Description	2
1.2	Java API for RESTful WSs: JAX-RS	3
1.3	Jersey	3
1.4	JSON and GSON	4
2	Programs presented at the lectures	5
2.1	Jersey-Rest-Plain	5
2.2	Jersey-Rest-Eclipse	6
2.3	REST-Test	7
2.4	Jersey-Rest-Client	7
3	Web Service authentication	8
3.1	HTTP Basic Authentication	8
3.2	Use of Keys	9
3.3	Third-Party Services	9
4	Server configuration	10
4.1	Initial management	10
4.2	Tomcat	11
4.3	Apache Server	12
4.4	Network management	13
5	Our program: RESTDict	14
5.1	Service Side	14
5.2	Client Side	15
5.3	Client Side - Graphics	16

1 Introduction

A Web Service is, in general, a service offered by a device to another device, which communicate with each other through the World Wide Web. It is a software running on a server, listening on specific ports for requests that arrive from clients.

When a request arrives, the Web Service answers that request with a response; for example it can answer with a web document (HTML, JSON, images, text..). Web Services are able to make available functionalities for other applications using the standard network. For an application to use a Web Service, it is only necessary to invoke it across the network.

During the lessons, we have seen two different types of Web Services: SOAP WS and REST WS. In this study we will see the advantages of using REST ones, and we will present examples of their usage.

1.1 REST Description

REST means Representational State Transfer. This is an architectural style, defined in 2000 by Roy Fielding, who is one of the main authors of the HTTP specifications. It represents a data transmission system based on HTTP.

The central element in a REST architecture is the **resource**. Everything that can be identified by its URI (for example, an URL) is a resource (documents, files..). A Web Service which uses this model is called a RESTful Web Service. It is important to notice that REST is not a protocol nor a specification: in fact, REST doesn't specify how the messages between clients and servers have to be sent; it defines guidelines and architectural constraints to follow.

REST architecture has some fundamental principles:

- The implementation is based on the principle of client-server separation (separation of concerns). This way it is possible, for example, to implement different user interfaces (clients) for different platforms.
- Data and functionalities are considered resources and are accessed by their Uniform Resource Identifiers (URIs).
- **Uniform interface:** the resources are accessed and manipulated by four simple rules: PUT, GET, POST, DELETE.
- **Statelessness:** the server should not store any session data of the client between requests. This means that every request should contain every information that the service needs to understand and satisfy the request. With this technique, it's as if every request from a client was always the first request from that client.
- **Scalability:** if a service receives a lot of requests, it can scale horizontally (creating more than one instance of that service) and use a *Load Balancer* to assign the incoming requests to the different instances. The stateless property removes the problem of sessions synchronization.
- Different representation of resources: every resource can have multiple representations, like text, HTML, JSON etc. A client can ask for a specific representation using the HTTP protocol (*Accept* attribute).

1.2 Java API for RESTful WSs: JAX-RS

JAX-RS is a Java API specification (JSR 370) designed to simplify the development of applications that use REST architecture. In order to do that, JAX-RS uses Java annotations: an annotation is a form of metadata that can be added to classic Java source code. POJO stands for *Plain Old Java Object*: it is a simple Java object. JAX-RS annotations are used to map a resource class (POJO) to a web resource. For example, the `@Path` annotation identifies a relative URI path. It can be a "static" string:

```
@Path("/dict")
public class Main {
```

Figure 1: Simple `@Path` Annotation

or it can be a URI path template, which is a URI with variables embedded within the syntax:

```
@Path("/{param}")
public Response getMsg(@PathParam("param") String msg)
```

Figure 2: Template `@Path` Annotation

In Figure 2 the template word in the URI is used as a parameter of the `getMsg` function. At runtime, the value of "param" is extracted from the URI, cast to `String` and then passed to the function as the parameter.

In JSR 370 we can find the main goals of the Java API:

- *POJO Based*: use of annotations, classes, interfaces to expose simple POJOs as Web resources.
- *HTTP Centric*: assume HTTP as the underlying protocol; mapping between HTTP and URIs and the corresponding API classes and annotations.
- *Format Independence*: applicable to a variety of HTTP content types. Possibility to add new ones.
- *Container Independence*: deployable in a variety of containers.
- *Inclusion in Java EE*: specify how to use Java EE within a Web resource class.

1.3 Jersey

Jersey is the reference implementation of the JAX-RS specification. It is an open source framework for developing RESTful Web Services in Java. It provides support for JAX-RS Java APIs.

The main idea is to give the developer the possibility to transform a simple POJO in a Web service, using runtime annotations. Jersey handles the HTTP requests and sends the request to the appropriate methods.

1.4 JSON and GSON

JSON (JavaScript Object Notation) is a very simple format for data exchange. It is completely independent from the programming language used.

It is composed of two main structures:

- **Object:** a JSON Object is like a JavaScript Object. It is composed of name/value pairs.
- **Array:** a JSON Array is an array of JSON Objects, all of them enclosed in square brackets and separated by a comma.

JSON supports different types: booleans (true/false), integer and floating point numbers, strings enclosed in double quotes.

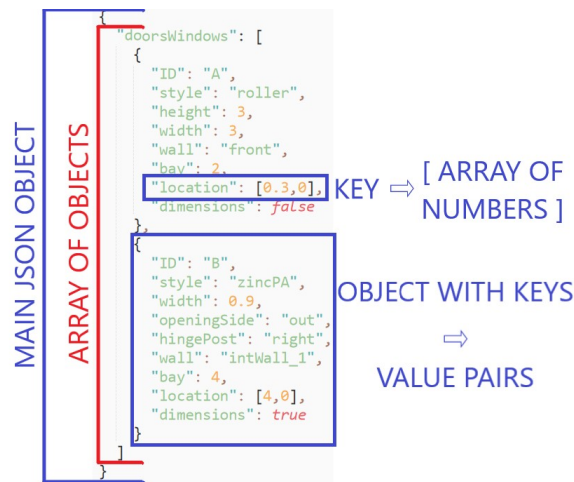


Figure 3: JSON Sample

GSON is a Java library developed by Google. It is useful to manipulate efficiently JSON Object and JSON Arrays in Java. It can parse a simple JSON string in Java Objects. It provides also functions to retransform Java Objects into JSON strings. For example:

```
class Word {  
    String english;  
    String italian;  
}
```

Figure 4: Java Class

```
String json = "[{\"english\":\"hello\", \"italian\": \"ciao\"}, {\"english\":\"love\", \"italian\": \"amore\"}]";  
Type wordListType = new TypeToken<ArrayList<Word>>().getType();  
ArrayList<Word> wordList = gson.fromJson(json, wordListType);  
String backToJson = gson.toJson(wordList, wordListType);
```

Figure 5: GSON utilities example

2 Programs presented at the lectures

During the lessons we have seen different programs about our topic. We used different approaches:

- **Services**

- *Tomcat*: prepare web.xml file in WEB-INF folder (and eventually all the necessary libraries). Compile the .java source code, put it in WEB-INF/classes folder, make a .war file of WEB-INF and META-INF. Deploy the .war file using Tomcat.
- *Eclipse IDE*: use of Eclipse EE Tools to create a Tomcat server directly into Eclipse and deploy the Web Service in there.

- **Clients**

- *CLI (Command Line Interface)*: compile .java files with command *javac*, and run it with *java* (including all necessary libraries with the option *-cp*).
- *Eclipse IDE*: build Java Projects and run programs directly in the IDE.

2.1 Jersey-Rest-Plain

This programs represents a simple REST Web Service that has different functionalities.

<pre>@Path("/mirror") public class SimpleService { @GET public Response getMsg() { String fixedMessage = "Fixed Message"; String output = "GET: Answer - " + fixedMessage ; return Response.status(200).entity(output).build(); } }</pre>	<pre>@GET @Path("/{param}") public Response getMsg(@PathParam("param") String msg) { String output = "GET with Param.: Answer - " + msg; return Response.status(200).entity(output).build(); }</pre>
(a)	(b)

Figure 6: GET functions

In the first line of the code we can see the `@Path` annotation: this Web Service will be accessed using the `/mirror` path.

The first two functions are two GET functions: when a client makes a GET request, those two will be used to build a response. In particular, the first function will be called when the request URI is only `/mirror`.

The second one will be called when after the word *mirror* there is another parameter. This is a template URI. The parameter of the URI is passed to the `getMsg` function and used to build a *dynamic* response, different from the first one which is *fixed*.

In the figure below we can see three POST functions implementation.

The two functions in figure (b) accept URIs with the specified parameters: *uploadtext* and *uploadjson*.

The one in (a) is called when the URI is */mirror/param*, with *param* a generic parameter different from the ones specified in the other two POST function. That parameter is passed to the *postMsg* function.

```

@POST
@Path("/{param}")
public Response postMsg(@PathParam("param") String msg) {
    String output = "POST with Param.: Answer - " + msg;
    return Response.status(200).entity(output).build();
}

```

(a)

```

@POST
@Path("/uploadtext")
@Consumes(MediaType.TEXT_PLAIN)
public Response postStrMsg(String msg) {
    String output = "POST with Uploaded String Body: Answer - " + msg;
    return Response.status(200).entity(output).build();
}

@POST
@Path("/uploadjson")
@Consumes(MediaType.APPLICATION_JSON)
public Response postJsonMsg(String msg) {
    String output = "POST with Uploaded JSON Body: Answer - " + msg;
    return Response.status(200).entity(output).build();
}

```

(b)

Figure 7: POST functions

In addition, we can note that the two POST functions in figure (b) contain the annotation `@Consumes`. This annotation is used to specify which MIME type of resource can this function accept by the client. The first function accepts text, the second one accepts JSON. When a client sends a request with a requested MIME type that can't be satisfied by no function of the Service, the Service responds with an HTTP 415 "Unsupported Media Type" error.

2.2 Jersey-Rest-Eclipse

This Service is really similar to the one mentioned above, except for the addition of the `@Produces` annotation.

```

@POST
@Path("/{param}")
@Produces(MediaType.TEXT_PLAIN)
public Response postMsg(@PathParam("param") String msg) {
    String output = "POST with Param.: Answer - " + msg;
    return Response.status(200).entity(output).build();
}

```

(a)

```

@POST
@Path("/uploadjson")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response postJsonMsg(String msg) {
    String output = "POST with Uploaded JSON Body: Answer - " + msg;
    return Response.status(200).entity(output).build();
}

```

(b)

Figure 8: `@Produces` and `@Consumes` annotations

This annotation is used to specify which is the MIME type of the response that the service sends to the client. In the figure below we can see two different `@Produces`, one for text and one for JSON. If there are no methods able to produce the MIME type requested by the client, an HTTP "406 Not Acceptable" error will be sent.

2.3 REST-Test

This Web Service differs from the others because it implements a POST method that receives JSON, manipulates it and resends back JSON to the client. It uses a simple class `Person`, and the GSON library.

```
class Person {
    String FirstName;
    String LastName;
    String Code;
}

@POST
@Path("/uploadjson")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public Response postJsonMsg(String msg) {
    Gson gson = new Gson();
    Person person = gson.fromJson(msg, Person.class);
    // Processing request
    person.Code = "1234";
    String backtojson = gson.toJson(person);
    return Response.status(200).entity(backtojson).build();
}
```

This programs reads JSON arriving from client and transforms it to a `Person` object using GSON library. It sets an attribute of that object and then it reuses GSON to send that object back to the client as a response.

2.4 Jersey-Rest-Client

This program implements a client which makes a request to the Web Service sending it a JSON representation of a `Person`. It connects to localhost, on port 8085 and it uses the path `/mirror/uploadjson`, defined in the programs above.

```
Gson gson = new Gson();
String jsonPerson = gson.toJson(person);
POST with JSON payload
Client client = ClientBuilder.newClient(new ClientConfig());
webTarget = client.target("http://localhost:8085/JR/rest").path("mirror/uploadjson");
invocationBuilder = webTarget.request(MediaType.APPLICATION_JSON);
response = invocationBuilder.post(Entity.entity(jsonPerson, MediaType.APPLICATION_JSON));
String downloadedJson = response.readEntity(String.class);

person = gson.fromJson(downloadedJson, Person.class);
System.out.print("Received: ");
System.out.println(person.FirstName + " " + person.LastName + " " + person.Code);
```

The response is formatted in JSON too: the client reads that response and uses GSON to retransform it to a `Person` object. It then prints its attributes to the standard output.

3 Web Service authentication

There are several ways for a client to use a service offered by a Web Service. Each developer uses a different *authentication* and *authorization* method, but all are variations of authentication methods to web services such as APIs. In general, there may be a difference between authentication and authorization, because some services can have different permissions for each feature. For example, a Web Service can have features delivered for free, while others provided only under payment. Other Web Services, on the other hand, all leave their own services with the same permissions: this makes no distinction between authentication and authorization to the individual service. The most commonly used methods for client-service authentication are:

- *HTTP Basic Authentication*
- *Use of Keys*
- *Third-Party Services*

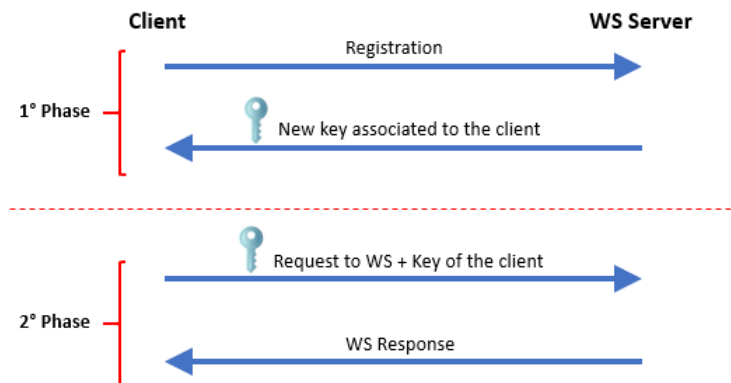


Figure 9: Key Exchange basic schema

3.1 HTTP Basic Authentication

HTTP Basic Authentication is a long-passed authentication method, that allows the client to authenticate by placing a username and password in the header of the HTTP request, as we can see in Figure 10.

```
1 GET / HTTP/1.1
2 Host: javaboss.it
3 Authorization: Basic Zm9vOmJhcg==
```

Figure 10: Sending information from the HTTP header

Both the username and password can be "encrypted" in Base64 by the client and then decrypted by the server. This method has been passed because sensitive data is sent and if username and password are chosen from the user they can be easy to guess.

3.2 Use of Keys

```
1 GET /something HTTP/1.1
2 X-API-Key: IP84UTvzJKds1Jomx8gIbTXcEEJSUilGqpxCmnnx
```

Figure 11: Sending the key to the server

Key Web Services have been introduced to eliminate the problem of exchanging easy-to-guess data. A key is a random sequence of characters that are difficult to memorize for a human. This key can be sent by the client to the HTTP request header or body, Figure 11. We have used this authentication method in our service.

3.3 Third-Party Services

Third-party services can also be used to allow authorization to your service. An example is OAuth, which for several years has guaranteed authentication to the service through an exchange of information between clients, servers and OAuth servers, as seen from the Figure 12.

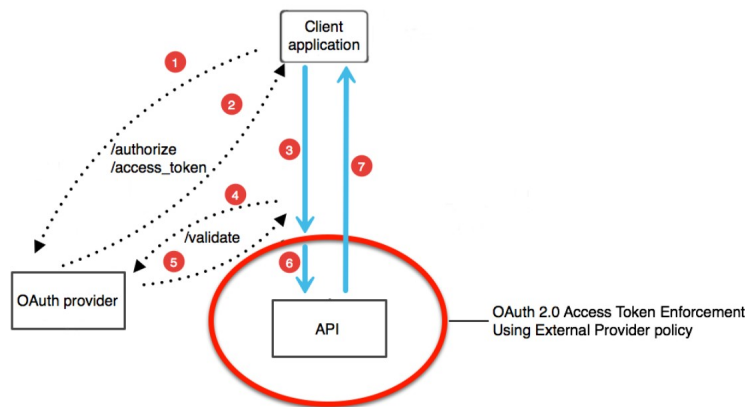


Figure 12: OAuth key

This type of method, of course, is slower than the others because it has to go through a server outside the Web Service server; at the same time, however, it ensures the security of the exchange of data with minimal effort.

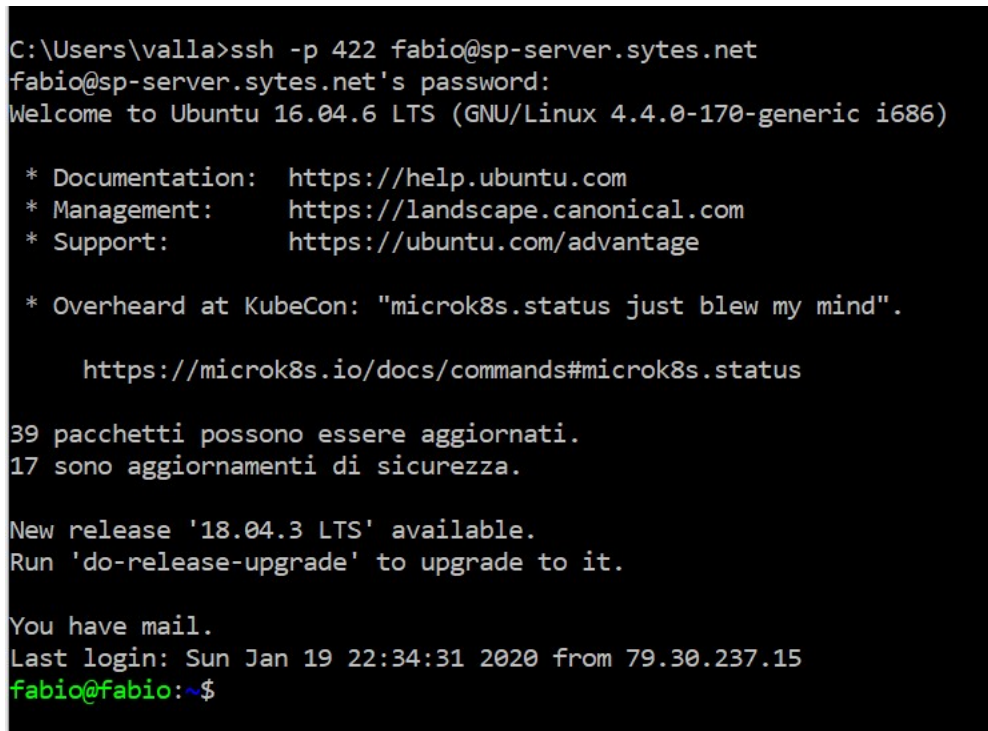
The implementations that have been listed are just some of the most commonly used implementations. Using stricter security protocols can improve the security of authentication to each Web Service. For example, the "Diffie-Hellman key exchange" algorithm allows you to create a pair of secure keys; the "RSA" algorithm allows you to create a public/private key pair. With RSA it is possible to encrypt the client request (or client credentials) with the public key, so only the server can decrypt the request.

4 Server configuration

For the configuration of our server we used a computer that we inserted into a home network. On the server we put *Ubuntu 16.04 LTS* on which we installed the *LAMP stack* and *Tomcat*. Since the server is part of a home network, we also managed port forwarding and Dynamic DNS.

4.1 Initial management

Once we installed Ubuntu on the machine, needing to connect even remotely, we installed the ssh server via the command `sudo apt-get install openssh-server`. Once ssh server is installed, we can connect to the server directly from the command line, as can be seen in Figure 13.

A terminal window showing an SSH connection from a Windows command prompt to a Linux server. The user 'valla' connects to 'fabio@sp-server.sytes.net' on port 422. The server is Ubuntu 16.04.6 LTS. The terminal output includes the Ubuntu welcome message, links for documentation, management, and support, a quote about KubeCon, a security update notification for 39 packages, and a new release '18.04.3 LTS' available. The session ends with the user 'fabio' at the prompt.

```
C:\Users\valla>ssh -p 422 fabio@sp-server.sytes.net
fabio@sp-server.sytes.net's password:
Welcome to Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-170-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

 * Overheard at KubeCon: "microk8s.status just blew my mind".

      https://microk8s.io/docs/commands#microk8s.status

39 pacchetti possono essere aggiornati.
17 sono aggiornamenti di sicurezza.

New release '18.04.3 LTS' available.
Run 'do-release-upgrade' to upgrade to it.

You have mail.
Last login: Sun Jan 19 22:34:31 2020 from 79.30.237.15
fabio@fabio:~$
```

Figure 13: ssh Connection

Having to work remotely, we also needed an ftp service to upload and download files. We, then, installed an ftp server and via an ftp client such as *FileZilla* we can connect to the server.

4.2 Tomcat

Tomcat was installed by downloading directly from the official website the latest released version, which in our case is the *9.0.30*. Once the package was downloaded and unpacked, we moved the folder to */opt/tomcat/* to keep this folder dedicated to Tomcat only.

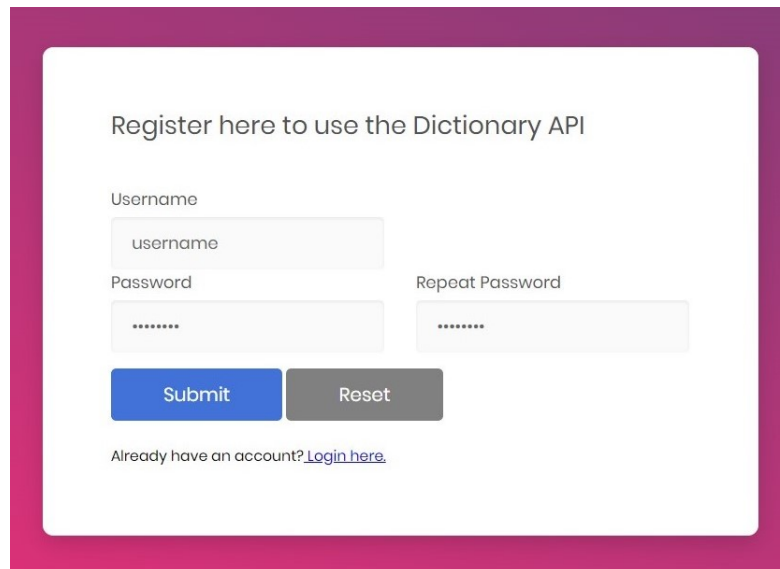
```
root@fabio:/# cd /opt/tomcat/apache-tomcat-9.0.30/bin/
root@fabio:/opt/tomcat/apache-tomcat-9.0.30/bin# ./startup.sh
Using CATALINA_BASE:   /opt/tomcat/apache-tomcat-9.0.30
Using CATALINA_HOME:   /opt/tomcat/apache-tomcat-9.0.30
Using CATALINA_TMPDIR: /opt/tomcat/apache-tomcat-9.0.30/temp
Using JRE_HOME:        /usr
Using CLASSPATH:       /opt/tomcat/apache-tomcat-9.0.30/bin/bootstrap.jar:/opt/tomcat/apache-tomcat-9.0.30/bin/tomcat-juli.jar
Tomcat started.
root@fabio:/opt/tomcat/apache-tomcat-9.0.30/bin#
```

Figure 14: Tomcat start command

With server started (Figure 14) and working correctly (Figure 15), we modified the file *tomcat-users.xml* within the *conf* folder, creating a user with *manager-gui* permission, so that we could enter the online space dedicated to webapps management.

Figure 15: Tomcat welcome page

4.3 Apache Server



Register here to use the Dictionary API

Username

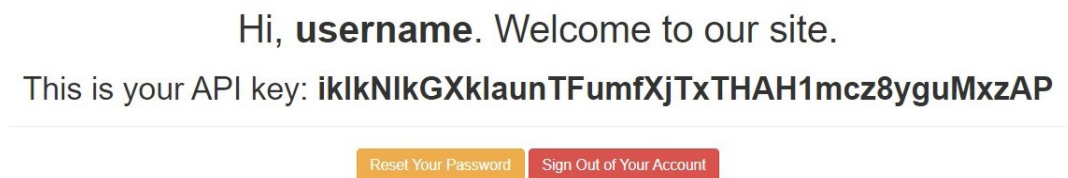
Password

Repeat Password

Already have an account? [Login here.](#)

Figure 16: Registration page

By installing the *LAMP stack*, we were able to use all the functions of *Apache Server*. We then created a simple registration form. When a user registers to the site, it gets a key (Figure 16).



Hi, **username**. Welcome to our site.

This is your API key: **iklkNlkGXklaunTFumfXjTxTHAH1mcz8yguMxzAP**

Figure 17: Welcome page

By accessing port *80*, in addition to registration, a user can login and see its key (Figure 17) or reset the account password. The registration form, of course, is not only facade, but associates each username with a key by saving the tuple in a MySQL database, in order to check if a user is qualified or not to use the Web Service.

4.4 Network management

To allow services to exit the home network in which they are located, it is necessary to change the rules within the router. The server gives a fixed local IP address and is placed in the demilitarized zone *DMZ*.

Port forwarding

- External port 422 mapped to internal 22 for ssh services.
- External port 421 mapped to internal 21 for ftp services.


#	Stato	Nome Servizio	IP sorgente	Interfaccia WAN	Indirizzo IP del Server	Porta Iniziale	Porta Finale	Porta di Traslazione Iniziale	Porta di Traslazione Finale
1		FTP SP-server		Predefinito	192.168.1.6	421	421	21	21
2		ssh per SP-server		Predefinito	192.168.1.6	422	422	22	22

Figure 18: Port forwarding

Dynamic DNS Service Use of NO-IP service to have a name for the website.

Every time the IP address changes, the Dynamic DNS sends the new IP to NO-IP in order to maintain the link with the site name.

5 Our program: RESTDict

Our program implements a RESTful sort of dictionary (english-italian).

The data exchange is realized in JSON, and the GSON library is used both in server and client source code in order to simplify JSON manipulation.

The program consists in four parts:

- *REST Dictionary - List*: WS that prints out all words in the dictionary. It is accessible only in GET method.
- *REST Dictionary - Translator*: the client sends a word in english and the WS answers with its italian translation. This feature is accessible both in GET and POST method.
- Implementation of a CLI client program that uses the *translation* functionality.
- Implementation of a graphic client program that uses the *translation* functionality.

5.1 Service Side

The service side is composed of two different Java classes: Dict and Main.

Dict class is the one which manages the import of the dictionary from a text file into a HashMap. Every time a client asks for a word, the service searches into the HashMap for that word. Every word is associated with a Word object (defined by us), which has two main String attributes: english and italian, which represent the same word in the two languages.

The other class is Main, which is the one which implements the service methods. Our service is accessible via the @Path */dict*. There are four methods implemented:

- **GET without parameter**: this function sends a welcome message and explains how to use the functionalities.
- **GET with parameter */dict-ls***: when the user inserts this parameter, the service answers with all the words contained in the dictionary with their translation.
- **GET with other parameter**: in this case, the *translation* functionality is activated. The service responds with the translation of the given word, or an error message if the word isn't in the dictionary.
- **POST with JSON payload**: this is the functionality used by out clients. This functions has the @Path annotation on */uploadjson*. In addition, it also has the annotation @Consumes that indicates that this functions accepts JSON strings as requests. It also has the @Produces annotation, to indicate that the response is in JSON too.

```
@POST
@Path("/uploadjson")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public Response postJsonMsg(String msg) throws FileNotFoundException, IOException {
```

Figure 19: Declaration of the postJsonMsg function

In this function, before searching the word in the dictionary, the service retrieves the key of the client and searches for it in the database. If the clients has sent a valid key, then it proceeds with the translation. If not, it sends an error message.

5.2 Client Side

The client side of our programs is composed of a simple Class which represents the REST client. The user has to insert a word as a command line arguments when executing the program from CLI. The program proceeds with a rapid check of the string inserted as argument (in particular, it only accepts letters), and then it builds a Word object with the *english* attribute set to the word passed as parameter.

The key needed for authentication is embedded in the client itself, and it is sent within the Word object to the server.

The client uses GSON library to transform Word Java object into a JSON string, and then it submits to the server a POST request with JSON payload, sending it to the following web target:

```
webTarget = client.target("http://sp-server.sytes.net:8080/RestDict").path("dict/uploadjson");
```

Then the client sends the request and listens for the response. It specifies both *accepted response MIME type*, with the function:

```
invocationBuilder = webTarget.request(MediaType.APPLICATION_JSON);
```

and the *MIME type of the content* of the request (also JSON):

```
response = invocationBuilder.post(Entity.entity(jsonWord, MediaType.APPLICATION_JSON));
```

Finally, when it has the response of the service, it prints on console the result of the research.

```
<terminated> RestDictClient (2) [Java Application]
Searching the word "love" in the database...

-----Search Results-----
love: amore
```


5.3 Client Side - Graphics

Once finished, the client program can be launched from the command line. To create an example more similar to real programs we decided to create a *.jar* executable with a graphical interface.

In order to realize the interface, we installed in eclipse the extension package *Window Builder*. It makes possible to build a window with buttons, labels and text fields (Figure 21).

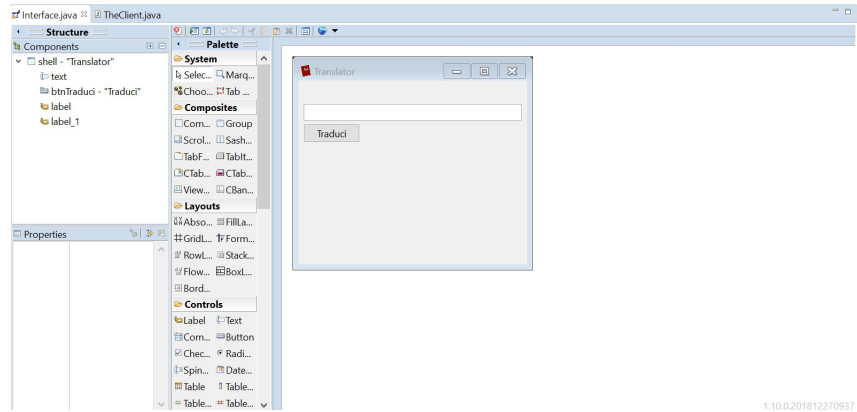


Figure 20: Graphic development

At the end, we exported the file as an *executable jar* and were able to run it on different machines. Subsequently, to make the program available to all Windows computers, we converted the *.jar* executable to a *.exe* executable via the *Launch4j* program.

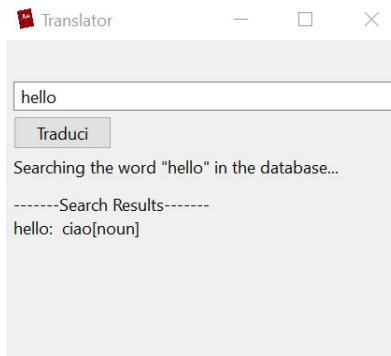


Figure 21: Translator client program

References

- [1] <https://docs.oracle.com/cd/E19798-01/821-1841/gipzh/index.html>
- [2] <https://docs.oracle.com/javaee/7/tutorial/jaxrs.htm>
- [3] https://2019.aulaweb.unige.it/pluginfile.php/173515/mod_resource/content/1/jaxrs-2_1-final-spec.pdf