Distributed Computing
Chapter 5 – REST Web Services

ITI 4 – INSA Rouen Normandie / 2020-2021

Cecilia ZANNI-MERK

What is REST?

- ***REST*** stands for **REpresentation State Transfer**, which requires clarification because the central abstraction in REST—the *resource*—does not occur in the acronym.
- A resource in the RESTful sense is anything that has an URI; that is, an identifier that satisfies formatting requirements.
  - The formatting requirements are what make URIs *uniform*.
  - Recall, too, that URI stands for Uniform *Resource* Identifier; hence, the notions of *URI* and *resource* are intertwined.
  - In practice, ***a resource is an informational item that has hyperlinks to it.***

What is REST?

- **RESTful web services** are lightweight, highly scalable and maintainable and are very commonly used to create APIs for web-based applications.

REST vs SOAP

- REST and SOAP are quite different.
- SOAP is a messaging protocol, whereas REST is a style of software architecture for distributed hypermedia systems -> the World Wide Web
- In the Web, HTTP is both a transport protocol and a messaging system because HTTP requests and responses are messages.
- The real data in HTTP messages can be conveyed using the MIME type system, and HTTP provides response status codes to inform the requester about whether a request succeeded and, if not, why.

More on resources

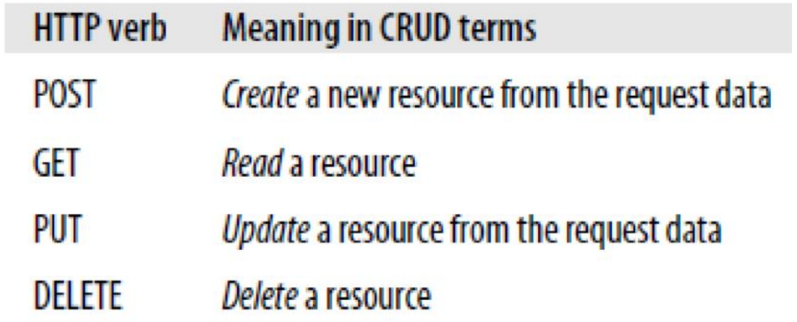
- As Web-based informational items, resources are without any interest unless they have at least one representation.
  - In the Web, representations are MIME-typed. The most common type of resource representation is probably still text/html, but nowadays resources tend to have multiple representations.
- Resources have state. A useful representation must capture a resource’s state.

More on resources

- In a RESTful request targeted at a resource, the resource itself remains on the service machine. The client typically receives a *representation* of the resource if the request succeeds.
    - It is the representation that transfers from the server machine to the client machine
- RESTful web services require not just resources to represent but also client-invoked operations on such resources. In other words, in a REST architecture, a REST Server simply provides access to resources and the REST client accesses and presents the resources.

More on resources

- Keep in mind that HTTP is an API and not simply a transport protocol, with its own *verbs* (also called *methods* or *CRUD operations*)



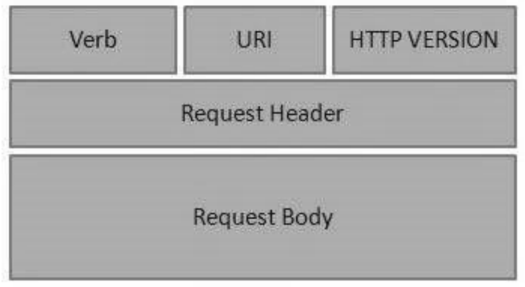
HTTP verb	Meaning in CRUD terms
POST	*Create* a new resource from the request data
GET	*Read* a resource
PUT	*Update* a resource from the request data
DELETE	*Delete* a resource

Representation of resources

- Once a resource is identified then its representation is to be decided using a standard format so that the server can send the resource in the above said format and client can understand the same format.
- REST does not impose any restriction on the format of a resource representation.
- Some important regarding the representation format of a resource in RESTful Web Services.
  - **Understandability**
  - **Completeness**
  - **Linkablity**

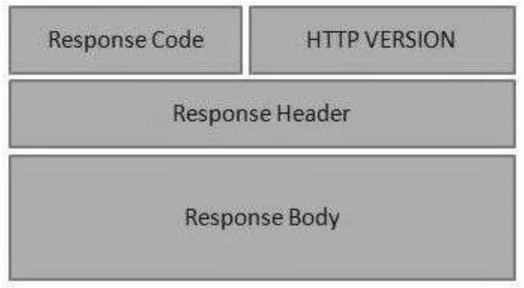
Messages

- HTTP Request



HTTP Request

- HTTP Response



HTTP Response

Addressing

- Addressing refers to locating a resource or multiple resources lying on the server.
- Each resource in REST architecture is identified by its URI that has the following format

```
<protocol>://<service-name>/<ResourceType>/<ResourceID>
```

- Constructing good URIs
  - Use plural nouns
  - Avoid using spaces
  - Use lowercase letters
  - Maintain backward compatibility
  - Use HTTP verbs

Addressing

- Addressing refers to locating a resource or multiple resources lying on the server.
- Each resource in REST architecture is identified by its URI that has the following format

```
<protocol>://<service-name>/<ResourceType>/<ResourceID>
```

- Constructing good URIs
  - Use plural nouns
  - Avoid using spaces
  - Use lowercase letters
  - Mainta
  - Use HT

```
http://localhost:8080/UserManagement/rest/UserService/getUser/1
http://localhost:8080/UserManagement/rest/UserService/users/1
```

Methods

- As we have discussed so far that RESTful web service makes heavy uses of HTTP verbs to determine the operation to be carried out on the specified resource(s).
- Here are important points to be considered:
  - GET operations are read only and are safe.
  - PUT and DELETE operations are idempotent means their result will always be the same no matter how many times these operations are invoked.
  - PUT and POST operations are nearly same with the difference lying only in the result where PUT operation is idempotent and POST operation can cause different results.

Statelessness

- A RESTful Web Service should not keep a client state on the server. This restriction is called **Statelessness**. It is the responsibility of the client to pass its context to the server.
- Advantages
  - Web services can treat each method request independently.
  - Web services need not maintain the client's previous interactions. It simplifies the application design.
  - As HTTP is itself a statelessness protocol, RESTful Web Services work seamlessly with the HTTP protocols.

Statelessness

- Disadvantages
  - Need of extra information in each request <- more bandwidth needed
  - Need to interpret the client's state if the client interactions are to be taken care of.

Security

- As RESTful Web Services work with HTTP URL Paths, it is very important to safeguard a RESTful Web Service in the same manner as a website is secured.
  - Validation
  - Session based authentication
  - Restriction on method execution
  - Validate malformed XML/JSON
  - Throw generic error messages

The HTTP codes

- When accessing a resource, a numeric code is received with the message
- HTTP codes are always three-digit and are categorized according to the number of hundreds.
  - 2xx indicates success.
  - 3xx redirects the client elsewhere.
  - 4xx indicates an error a client error
  - 5xx indicates a server error.

The HTTP codes

- When accessing a resou with the message
- HTTP codes are always according to the numbe
  - 2xx indicates success.
  - 3xx redirects the client
  - 4xx indicates an error a
  - 5xx indicates a server er

Depending on the HTTP code received, the client application may decide what to do next. For example, if a server responds with a code 500, the client will not be able to assign the anticipated data to a variable. On the other hand with a code 200, the client will know that the answer was good and that he will be able to proceed to the interpretation of the received information !

More on HTTP codes

- `200  OK`
- `201  CREATED`
- `204  NO CONTENT`
- `304  NOT MODIFIED`
- `400  BAD REQUEST`
- `401  UNAUTHORIZED`
- `403  FORBIDDEN`
- `404  NOT FOUND`
- `409  CONFLICT`
- `500  INTERNAL SERVER ERROR`

Summary of the RESTful Features

- In a request, the pairing of an HTTP verb such as GET with a URI such as http://.../users specifies a CRUD operation against a resource; in this example, a request to read available information about the users.
- The service uses HTTP status codes such as 404 (*resource not found*) and 405 (*method not allowed*) to respond to bad requests.
- If the request is a good one, the service responds with an XML representation that captures the state of the requested resource.

Summary of the RESTful Features

- The service can take advantage of MIME types. A client can issue a request indicating a preference for the type of representation returned (for instance, text/plain as opposed to text/xml or text/html).
- The RESTful service implementation is not constrained in the same way as a SOAP based service precisely because there is no formal service contract.
  - The implementation is flexible but, of course, likewise ad hoc.

Java Clients Against Real-World RESTful Services

- First, we need a tool for HTTP request transmission and analysis
  - There are several plugins for Firefox or Chrome … for example RESTClient, a debugger for RESTful web services.
  - Curl (https://curl.haxx.se/) , a command line tool and library for transferring data with URLs

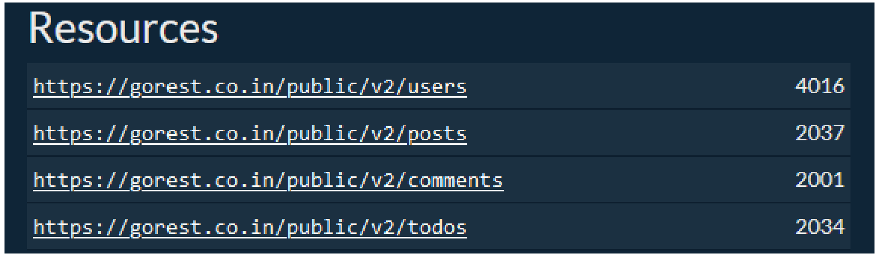
GoRest

- GoRest proposes an online **free** REST API for Testing and Prototyping Web and Android applications
- They also provide an online REST console for rapid testing
- However, signing up is necessary to get an “authentication token” to be used in the queries

https://gorest.co.in/

GoRest

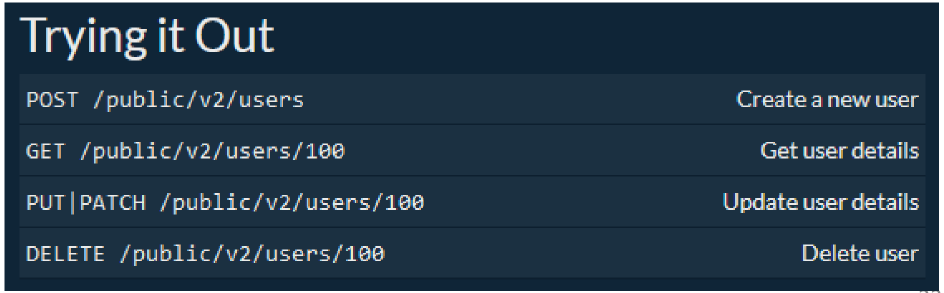
- They propose several resources



Resources

https://gorest.co.in/public/v2/users	4016
https://gorest.co.in/public/v2/posts	2037
https://gorest.co.in/public/v2/comments	2001
https://gorest.co.in/public/v2/todos	2034

- And support all the HTTP verbs



Trying it Out

`POST /public/v2/users`	Create a new user
`GET /public/v2/users/100`	Get user details
`PUT|PATCH /public/v2/users/100`	Update user details
`DELETE /public/v2/users/100`	Delete user

Up to you!!

- Familiarize yourself with the GoRest console on the website
  - Retrieve all the users that are called Victor
  - Modify user 1622’s name to “ZZZ Johnston”

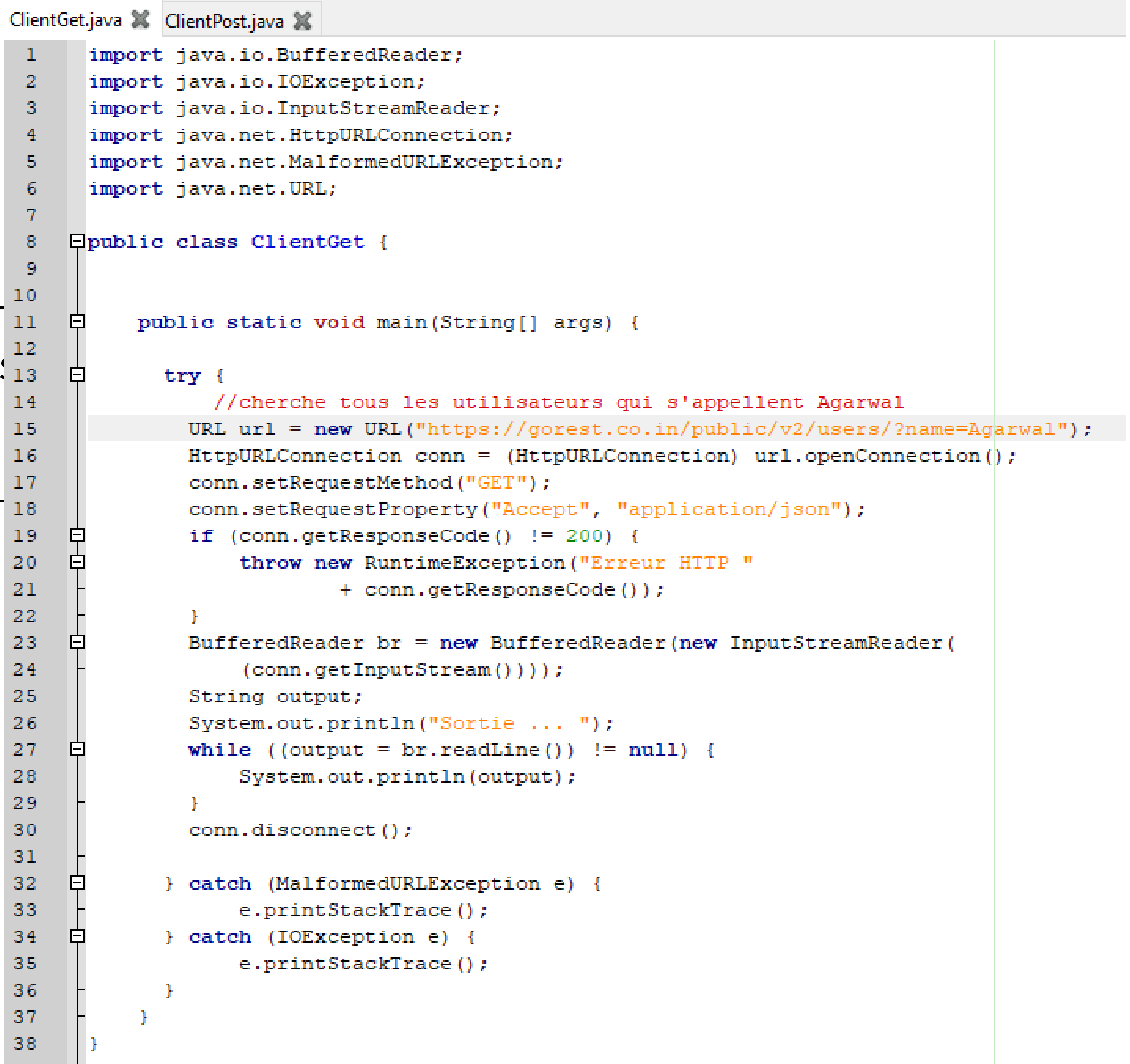
- There are also some plugins for Firefox or other browsers that allow to test REST queries
  - See, for example, ***RESTClient***
  - Reproduce the two proposed queries

A first Java client

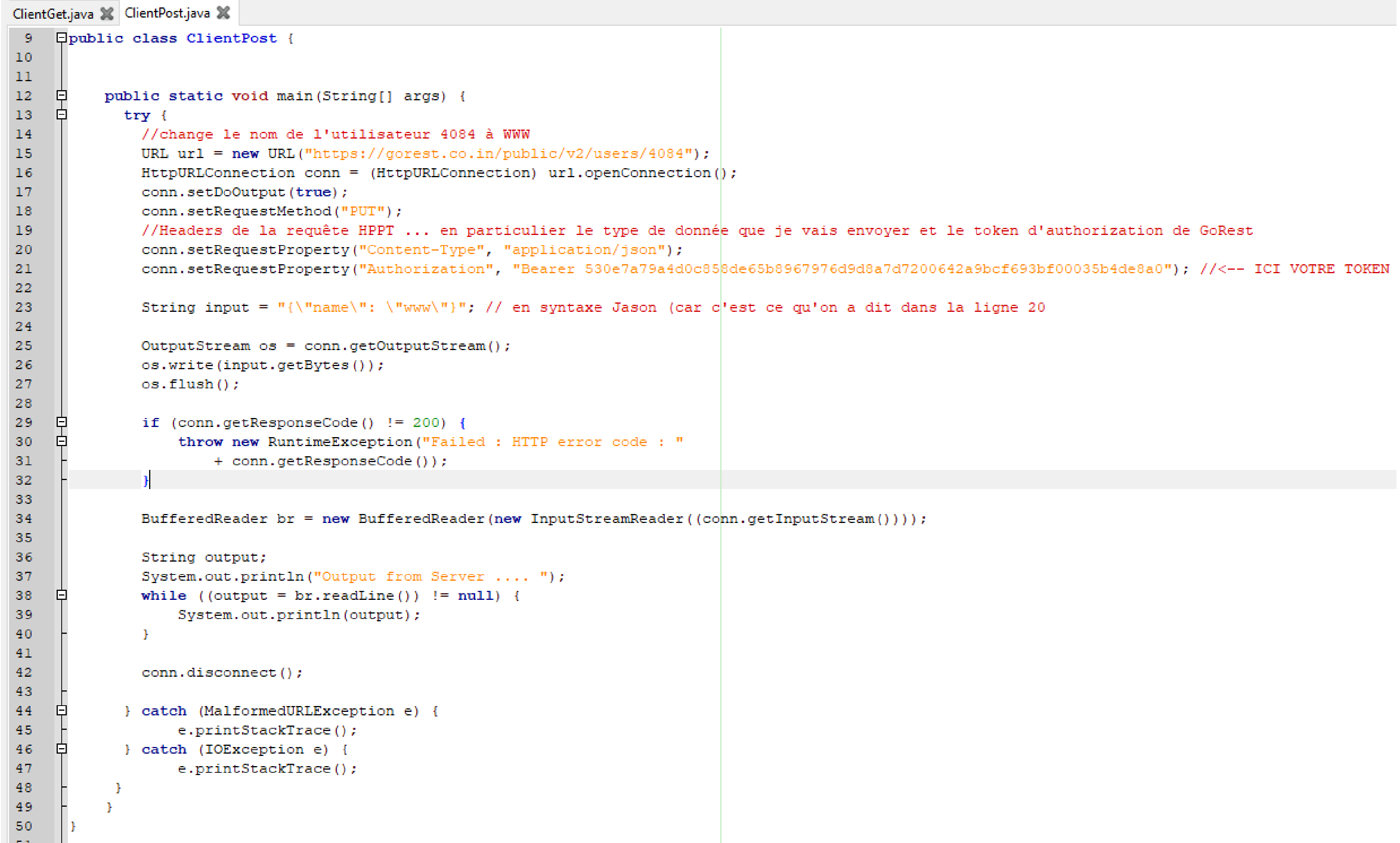
- Using integrated libraries for HTTP clients, to send GET and POST requests
  - `java.net.URL`
  - `java.net.HttpURLConnection`

A first Java client

- Using integrated li
  GET and POST reque
  - java.net.URL
  - java.net.HttpURL



```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.HttpURLConnection;
import java.net.MalformedURLException;
import java.net.URL;

public class ClientGet {


    public static void main(String[] args) {

      try {
          //cherche tous les utilisateurs qui s'appellent Agarwal
        URL url = new URL("https://gorest.co.in/public/v2/users/?name=Agarwal");
        HttpURLConnection conn = (HttpURLConnection) url.openConnection();
        conn.setRequestMethod("GET");
        conn.setRequestProperty("Accept", "application/json");
        if (conn.getResponseCode() != 200) {
            throw new RuntimeException("Erreur HTTP "
                    + conn.getResponseCode());
        }
        BufferedReader br = new BufferedReader(new InputStreamReader(
            (conn.getInputStream())));
        String output;
        System.out.println("Sortie ... ");
        while ((output = br.readLine()) != null) {
            System.out.println(output);
        }
        conn.disconnect();

      } catch (MalformedURLException e) {
            e.printStackTrace();
      } catch (IOException e) {
            e.printStackTrace();
      }
    }
}
```



ClientGet.java ClientPost.java

```
 9  public class ClientPost {
10
11
12      public static void main(String[] args) {
13        try {
14          //change le nom de l'utilisateur 4084 à WWW
15          URL url = new URL("https://gorest.co.in/public/v2/users/4084");
16          HttpURLConnection conn = (HttpURLConnection) url.openConnection();
17          conn.setDoOutput(true);
18          conn.setRequestMethod("PUT");
19          //Headers de la requête HPPT ... en particulier le type de donnée que je vais envoyer et le token d'authorization de GoRest
20          conn.setRequestProperty("Content-Type", "application/json");
21          conn.setRequestProperty("Authorization", "Bearer 530e7a79a4d0c858de65b8967976d9d8a7d7200642a9bcf693bf00035b4de8a0"); //<-- ICI VOTRE TOKEN
22
23          String input = "{\"name\": \"www\"}"; // en syntaxe Jason (car c'est ce qu'on a dit dans la ligne 20
24
25          OutputStream os = conn.getOutputStream();
26          os.write(input.getBytes());
27          os.flush();
28
29          if (conn.getResponseCode() != 200) {
30              throw new RuntimeException("Failed : HTTP error code : "
31                  + conn.getResponseCode());
32          }
33
34          BufferedReader br = new BufferedReader(new InputStreamReader((conn.getInputStream())));
35
36          String output;
37          System.out.println("Output from Server .... ");
38          while ((output = br.readLine()) != null) {
39              System.out.println(output);
40          }
41
42          conn.disconnect();
43
44        } catch (MalformedURLException e) {
45              e.printStackTrace();
46        } catch (IOException e) {
47              e.printStackTrace();
48       }
49      }
50  }
```

Other available APIs to explore

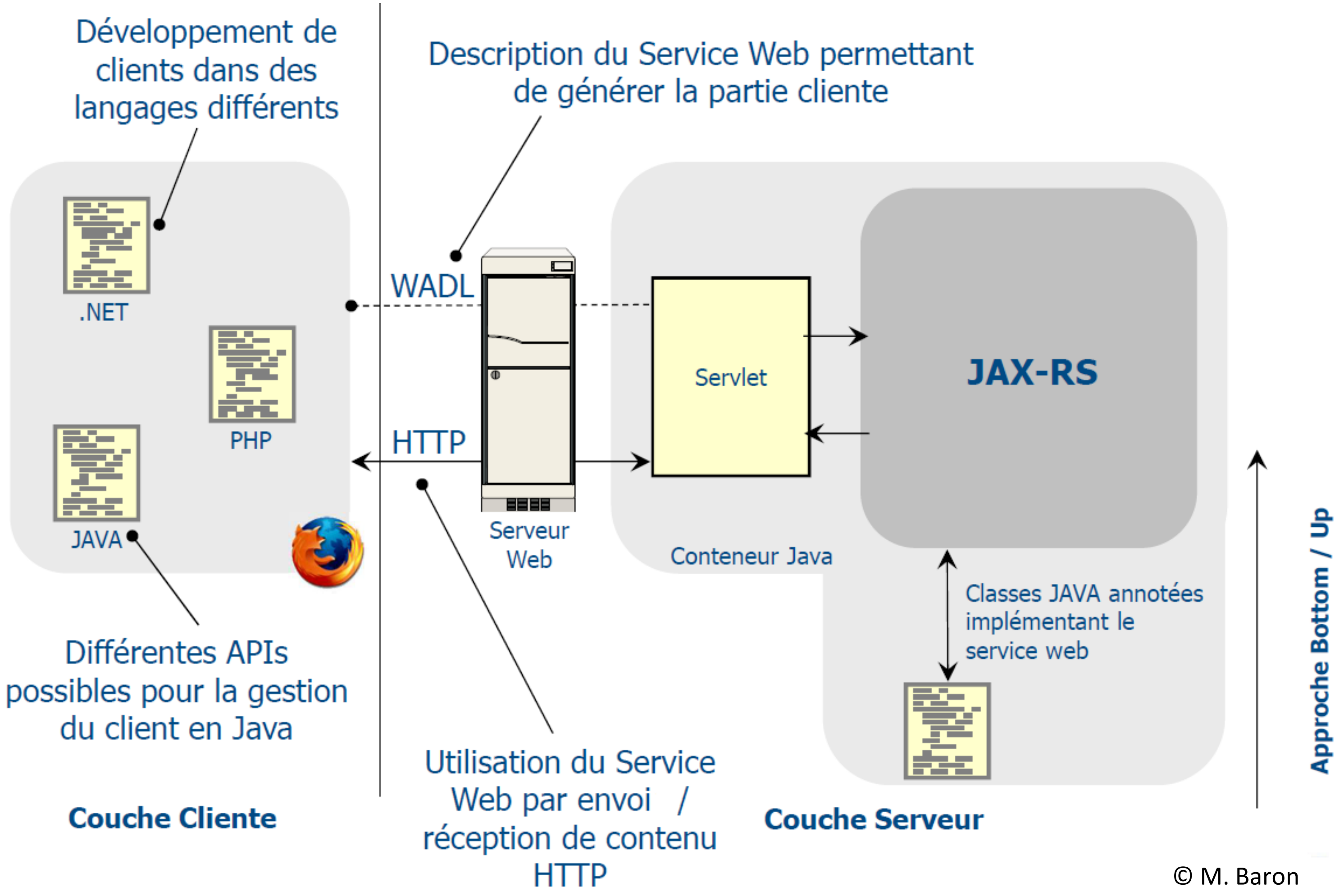
- Gmail https://developers.google.com/gmail/api/
- OpenStreetMaps https://wiki.openstreetmap.org/wiki/API

A more sophisticated way: JAX-RS

- JAX-RS : Java API for RESTful Web Services
- JAX-RS is integrated into Java since Java EE 6
- The development of REST Web Services with JAX-RS is based on Java annotations
- Several implementations exist: ***JERSEY (Oracle, that is the reference),*** CXF (Apache), RESTEasy (JBoss/WildFly), RESTlet
- The specification only describes the server part, the client part depends on each implementation

A more sophisticated way: JAX-RS

- The JAX-RS API uses Java annotations to simplify the development of RESTful web services.
- Developers decorate Java class files with JAX-RS annotations to define resources and the actions that can be performed on those resources.
- JAX-RS annotations are runtime annotations; therefore, runtime reflection will generate the helper classes and artifacts for the resource.
- A Java EE application archive containing JAX-RS resource classes will have the resources configured, the helper classes and artifacts generated, and the resource exposed to clients by deploying the archive to a Java EE server.


Développement de clients dans des langages différents
Description du Service Web permettant de générer la partie cliente
.NET
PHP
JAVA
WADL
HTTP
Serveur Web
Servlet
JAX-RS
Conteneur Java
Classes JAVA annotées implémentant le service web
Différentes APIs possibles pour la gestion du client en Java
Utilisation du Service Web par envoi / réception de contenu HTTP
Couche Cliente
Couche Serveur
Approche Bottom / Up

Development

- Web Services development with JAX-RS is based on POJOs using JAX-RS specific annotations
- No description required in configuration files
- Only the configuration of a **JAX-RS Servlet** is required to perform the bridge between HTTP requests and annotated Java classes
- A REST Web Service is deployed in a Web application

A word about servlets and containers

- A Java servlet is a Java program that extends the capabilities of a server. Although servlets can respond to any types of requests, they most commonly implement applications hosted on Web servers
- To deploy and run a servlet, a *web container* (also known as a *servlet container*) must be used.
- A web container is essentially the component of a web server that interacts with the servlets.
- It is responsible for managing the lifecycle of servlets, mapping a URL to a particular servlet and ensuring that the URL requester has the correct access rights.

A word about servlets and containers

- Apache Tomcat is the reference implementation for servlet containers (http://tomcat.apache.org)
  - TOMCAT_HOME/bin, contains startup and shutdown scripts
  - TOMCAT_HOME/logs, contains logs to allow monitoring and diagnosis
  - TOMCAT_HOME/webapps, servlets are deployed as WAR (Web ARchive) files, which are JAR files with a .war extension in this folder.
- Almost every production-grade servlet has a configuration file named **web.xml**, that generally needs to be edited
- The servlet needs to be packaged (as a .war file) and deployed in the TOMCAT_HOME/webapps folder

You can use WildFly as a servlet container if you feel more comfortable with it

Development

- Unlike SOAP Web Services there is no possibility to develop a REST service from the service description file
- Only a Bottom / Up approach is available
  - Create and annotate a POJO
  - Compile, Deploy and Test
- Possibility to access the WADL document
  - The WADL description file is automatically generated by JAX-RS
- It is possible to use the WADL to generate the client
  - However, most of the frameworks provide APIs to generate the client

Annotation	Description
@Path	The @Path annotation's value is a relative URI path indicating where the Java class will be hosted: for example, /helloworld. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: /helloworld/{username}.
@GET	The @GET annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@POST	The @POST annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PUT	The @PUT annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@DELETE	The @DELETE annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@HEAD	The @HEAD annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PathParam	The @PathParam annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation.
@QueryParam	The @QueryParam annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
@Consumes	The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
@Produces	The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, "text/plain".
@Provider	The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such as MessageBodyReader and MessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity and that can be built using Response.ResponseBuilder.

Annotation	Description
@Path	The @Path annotation's value is a relative URI path indicating where the Java class will be hosted: for example, /helloworld. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: /helloworld/{username}.
@GET	The @GET annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@POST	The @POST annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PUT	The @PUT annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@DELETE	The @DELETE annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@HEAD	The @HEAD annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
@PathParam	The @PathParam annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation.
@QueryParam	The @QueryParam annotation is a type c request URI query parameters.
@Consumes	The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
@Produces	The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, "text/plain".
@Provider	The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such as MessageBodyReader and MessageBodyWriter. For HTTP requests, the MessageBodyReader is used to map an HTTP request entity body to method parameters. On the response side, a return value is mapped to an HTTP response entity body by using a MessageBodyWriter. If the application needs to supply additional metadata, such as HTTP headers or a different status code, a method can return a Response that wraps the entity and that can be built using Response.ResponseBuilder.

http://docs.oracle.com/javaee/7/api/.

Most commonly used annotations

- `@Path` Relative path of the resource class/method.
- `@GET` HTTP Get request, used to fetch resource.
- `@PUT` HTTP PUT request, used to create resource.
- `@POST` HTTP POST request, used to create/update resource.
- `@DELETE` HTTP DELETE request, used to delete resource.
- `@HEAD` HTTP HEAD request, used to get status of method availability.

Most commonly used annotations

- `@Produces` States the HTTP Response generated by web service.
  - For example, APPLICATION/XML, TEXT/HTML, APPLICATION/JSON etc.
- `@Consumes` States the HTTP Request type.
  - For example, `application/x-www-formurlencoded` to accept form data in HTTP body during POST request.
- `@PathParam` Binds the parameter passed to the method to a value in path.
- `@QueryParam` Binds the parameter passed to method to a query parameter in the path.

Most commonly used annotations

- `@MatrixParam` Binds the parameter passed to the method to a HTTP matrix parameter in path.
- `@HeaderParam` Binds the parameter passed to the method to a HTTP header.
- `@CookieParam` Binds the parameter passed to the method to a Cookie.
- `@FormParam` Binds the parameter passed to the method to a form value.
- `@DefaultValue` Assigns a default value to a parameter passed to the method.
- `@Context` Context of the resource.
    - For example, `HTTPRequest` as a context.

Two main concepts

- Root resource classes
  - Are POJOs
  - Are either annotated with `@Path`
  - Or have at least one method annotated with `@Path`
  - Or have a request method designator, such as `@GET`, `@PUT`, `@POST` or `@DELETE`.
- Resource methods
  - Are methods of a resource class
  - Are annotated with a request method designator

Jersey

- Jersey is the centerpiece project for *JAX-RS* (Java API for XML-RESTful Web Services).
- Jersey applications can be deployed through familiar commercial-grade containers such as standalone Tomcat, but Jersey also provides the lightweight Grizzly container that is well suited for learning the framework. It also works well with Maven.
- A deployed Jersey service automatically generates a WADL, which is then available through a standard GET request.

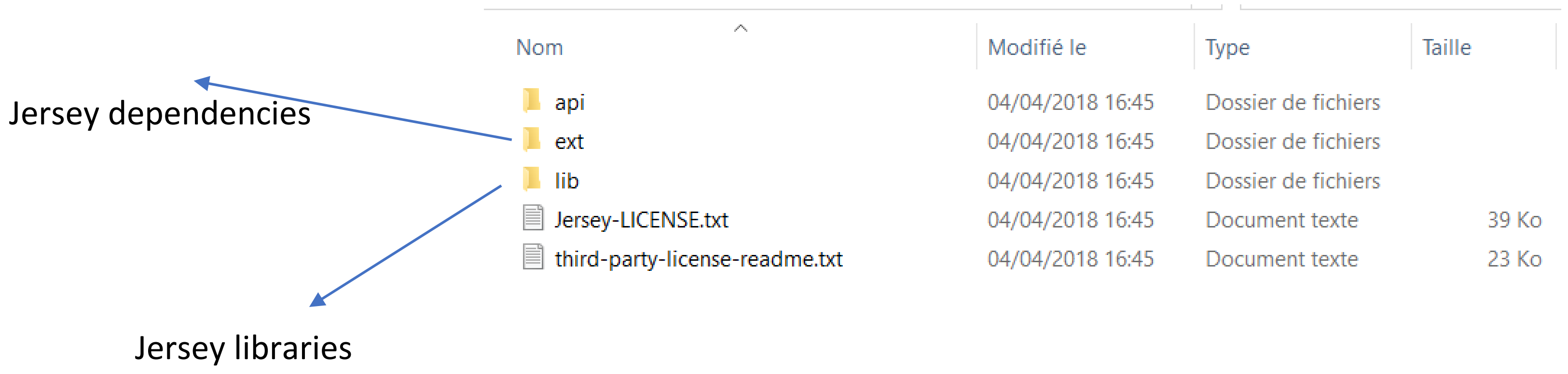
Jersey

- A Jersey service adheres to the REST principles.
- A service accepts the usual RESTful requests for CRUD operations specified with the standard HTTP verbs GET, POST, DELETE, and PUT.
- A request is targeted at a Jersey resource, which is a POJO.

- A good place to start is https://docs.oracle.com/cd/E19776-01/820-4867/ggnyk/index.html

Implementation: Environment Setup

- Be sure you have **Eclipse IDE for Enterprise Java Developers** installed
- Setup the Jersey Framework Libraries
  - Download the **Jersey 2.35 (implementing JAX-RS 2.1)** framework binaries from the following link – https://eclipse-ee4j.github.io/jersey/
  - Decompress and check that you have the following folder structure



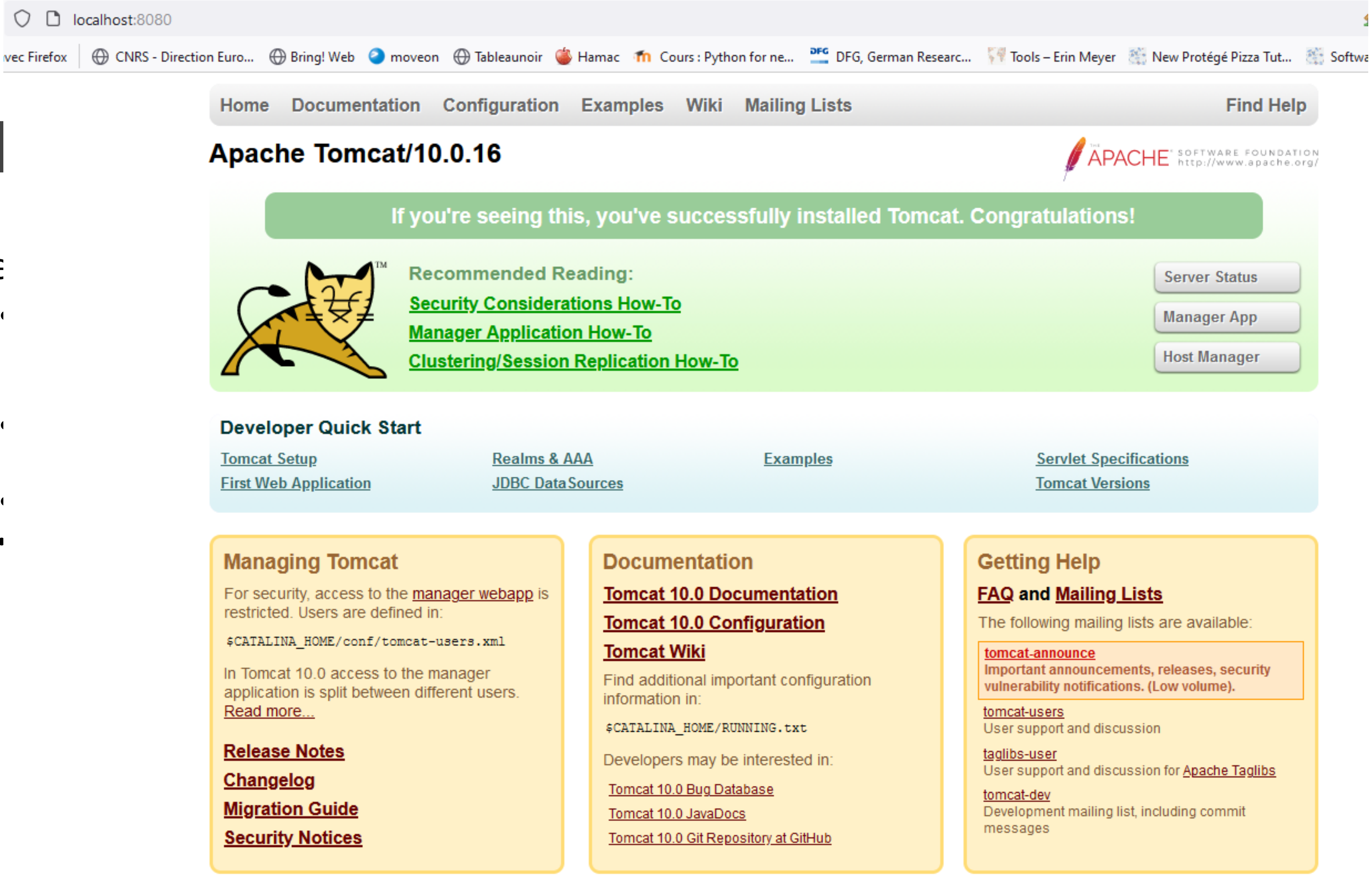
Nom	Modifié le	Type	Taille
api	04/04/2018 16:45	Dossier de fichiers	
ext	04/04/2018 16:45	Dossier de fichiers	
lib	04/04/2018 16:45	Dossier de fichiers	
Jersey-LICENSE.txt	04/04/2018 16:45	Document texte	39 Ko
third-party-license-readme.txt	04/04/2018 16:45	Document texte	23 Ko

Implementation: Environment Setup

- Setup Apache Tomcat
  - Download **Tomcat versions 9 or 10** from https://tomcat.apache.org/. Once you downloaded the installation, unpack the binary distribution into a convenient location.
  - Set the `CATALINA_HOME` environment variable pointing to the installation location.
  - The `/bin` folder contains scripts for launching and stopping Tomcat
  - After a successful startup, the default web applications included with Tomcat will be available by visiting **http://localhost:8080/**.

Im

- Se



localhost:8080

vec Firefox | CNRS - Direction Euro... | Bring! Web | moveon | Tableaunoir | Hamac | Cours : Python for ne... | DFG, German Researc... | Tools – Erin Meyer | New Protégé Pizza Tut... | Softwa

Home Documentation Configuration Examples Wiki Mailing Lists Find Help

Apache Tomcat/10.0.16

APACHE SOFTWARE FOUNDATION http://www.apache.org/

If you're seeing this, you've successfully installed Tomcat. Congratulations!

Recommended Reading:

Security Considerations How-To

Manager Application How-To

Clustering/Session Replication How-To

Server Status

Manager App

Host Manager

Developer Quick Start

Tomcat Setup	Realms & AAA	Examples	Servlet Specifications
First Web Application	JDBC DataSources		Tomcat Versions

Managing Tomcat

For security, access to the manager webapp is restricted. Users are defined in:

```
$CATALINA_HOME/conf/tomcat-users.xml
```

In Tomcat 10.0 access to the manager application is split between different users. Read more...

Release Notes

Changelog

Migration Guide

Security Notices

Documentation

Tomcat 10.0 Documentation

Tomcat 10.0 Configuration

Tomcat Wiki

Find additional important configuration information in:

```
$CATALINA_HOME/RUNNING.txt
```

Developers may be interested in:

Tomcat 10.0 Bug Database

Tomcat 10.0 JavaDocs

Tomcat 10.0 Git Repository at GitHub

Getting Help

FAQ and Mailing Lists

The following mailing lists are available:

tomcat-announce
Important announcements, releases, security vulnerability notifications. (Low volume).

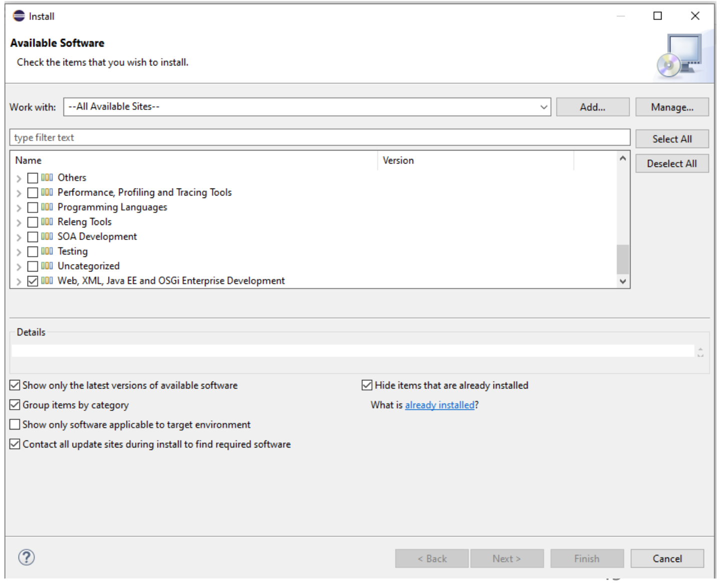
tomcat-users
User support and discussion

taglibs-user
User support and discussion for Apache Taglibs

tomcat-dev
Development mailing list, including commit messages

A first example

- The first step is to create a Dynamic Web Project using Eclipse IDE. Follow the option **File → New → Other → Web → Dynamic Web Project** wizard from the wizard list.
  - If you don't have "Dynamic Web Project" you need to update your Eclipse.
  - Go to **Help → Install new software**, choose "All available sites" in *Work with* and download "Web, XML, Java EE and OSGi Enterprise Development"
- Create your project and call it "Hello"



Install

Available Software

Check the items that you wish to install.

Work with: --All Available Sites--

type filter text

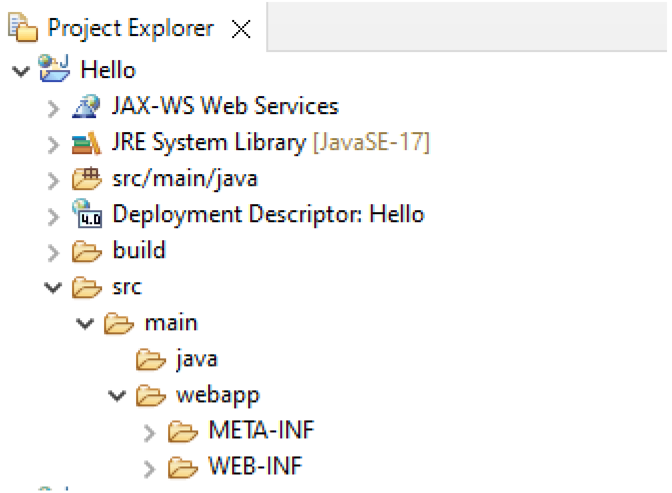
Name	Version
☐ Others	
☐ Performance, Profiling and Tracing Tools	
☐ Programming Languages	
☐ Releng Tools	
☐ SOA Development	
☐ Testing	
☐ Uncategorized	
☑ Web, XML, Java EE and OSGi Enterprise Development	

Details

☑ Show only the latest versions of available software
☑ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

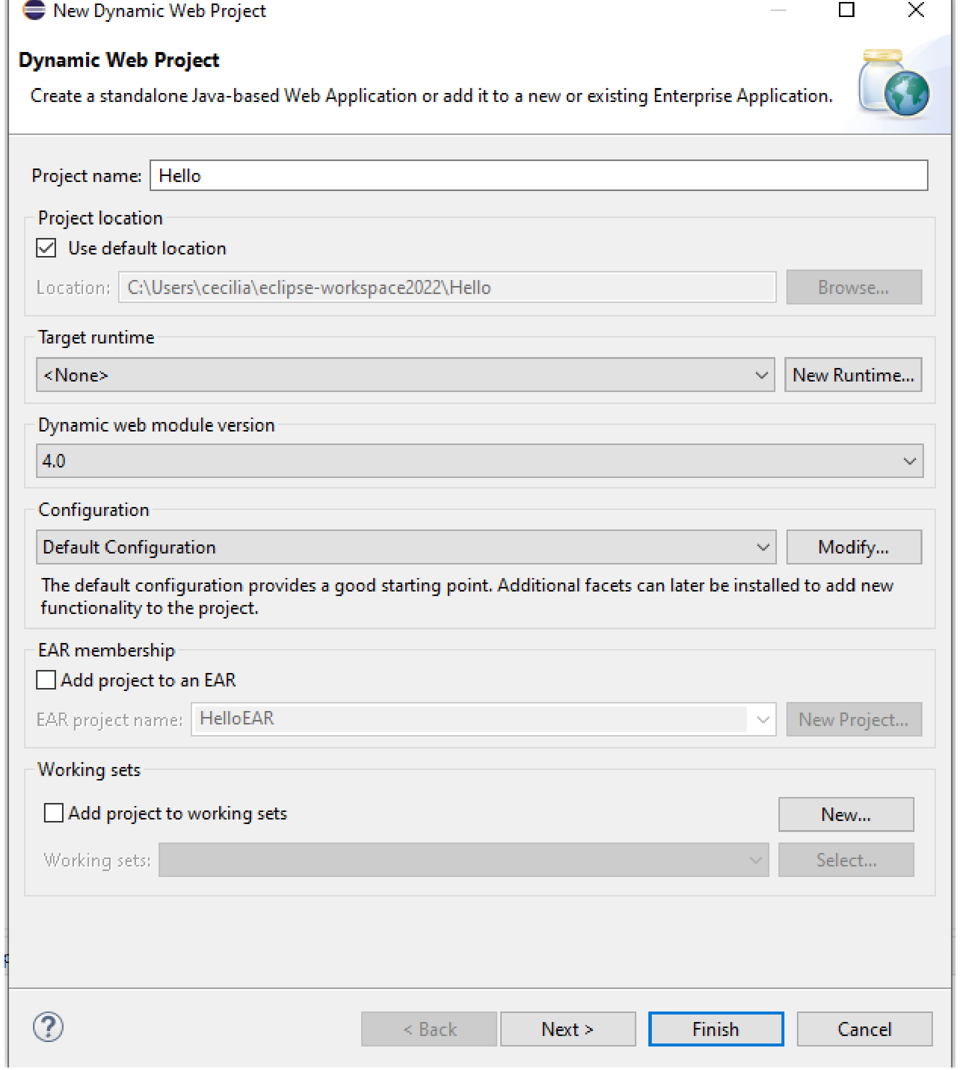
A first example

- Your project structure should look similar to this



Project Explorer

- Hello
  - JAX-WS Web Services
  - JRE System Library [JavaSE-17]
  - src/main/java
  - Deployment Descriptor: Hello
  - build
  - src
    - main
      - java
      - webapp
        - META-INF
        - WEB-INF



New Dynamic Web Project

Dynamic Web Project

Create a standalone Java-based Web Application or add it to a new or existing Enterprise Application.

Project name: Hello

Project location

☑ Use default location

Location: C:\Users\cecilia\eclipse-workspace2022\Hello [Browse...]

Target runtime

<None> [New Runtime...]

Dynamic web module version

4.0

Configuration

Default Configuration [Modify...]

The default configuration provides a good starting point. Additional facets can later be installed to add new functionality to the project.

EAR membership

☐ Add project to an EAR

EAR project name: HelloEAR [New Project...]

Working sets

☐ Add project to working sets [New...]

Working sets: [Select...]

[< Back] [Next >] [Finish] [Cancel]

Adding the required libraries

- As a second step let us add Jersey Framework and its dependencies (libraries) in our project. Copy ***all the .jars*** of the following directories of the downloaded jersey zip folder in the **src/main/webapp**/**WEB-INF/lib** directory of the project (do not copy the folders, only the ***.jars***).
  - \jaxrsFOLDER\jaxrs-ri\api
  - \jaxrsFOLDER\jaxrs-ri\ext
  - \jaxrsFOLDER\jaxrs-ri\lib

Do not copy the folders, only the ***.jars***

Adding the required libraries

- Now, right click on your project name **Hello** and then follow the option available in the context menu – **Build Path → Configure Build Path** to display the Java Build Path window.

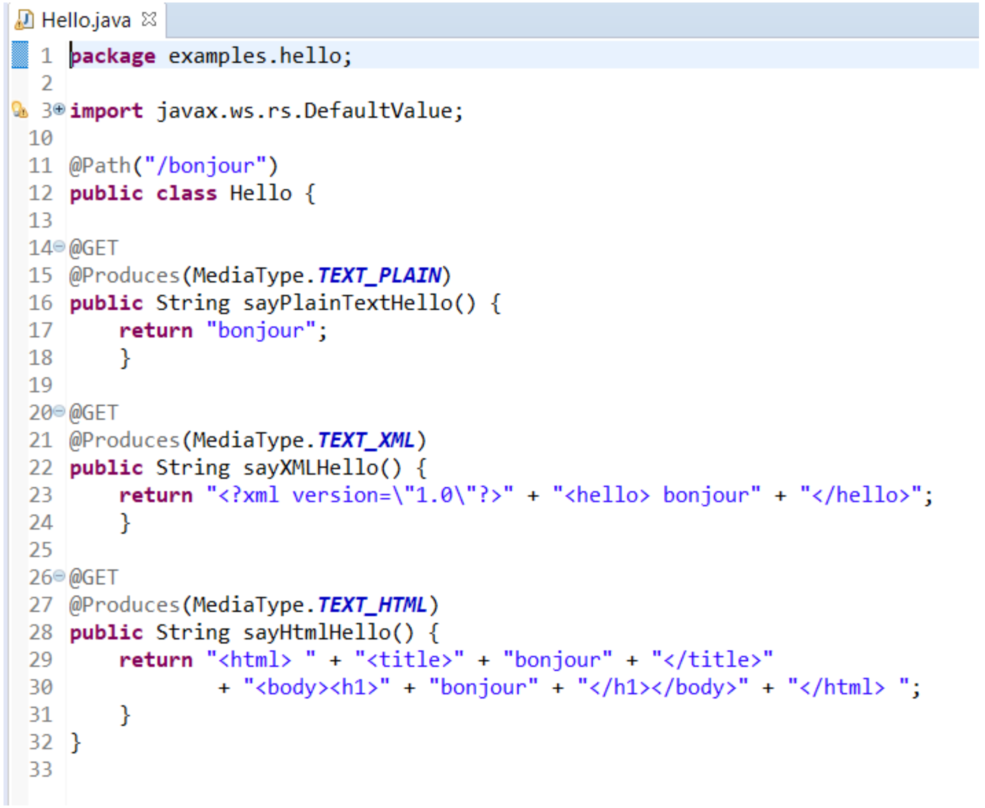
- Finally use the **Add External JARs** button available under **Libraries** tab (at the level of Classpath) to add the JARs present in `WEB-INF/lib` directory.

Creating the source files

- Create a new package `examples.hello`
- Create the class `Hello.java` in the package.
    - You can download it from moodle
    - Remark that it is a POJO
- There are some important points to be noted about the main program, `Hello.java`
    - The JAX-RS annotations ensure the connection between the POJO class and the HTTP request or response

Creating the source files

- The first step is to specify a path for the web service using the `@Path` annotation. The URI for requests ends with “bonjour”
- `@GET` specifies that the method answers to a GET request.
- `@Produces` specifies the type of the response


Hello.java

```java
package examples.hello;

import javax.ws.rs.DefaultValue;

@Path("/bonjour")
public class Hello {

@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello() {
    return "bonjour";
    }

@GET
@Produces(MediaType.TEXT_XML)
public String sayXMLHello() {
    return "<?xml version=\"1.0\"?>" + "<hello> bonjour" + "</hello>";
    }

@GET
@Produces(MediaType.TEXT_HTML)
public String sayHtmlHello() {
    return "<html> " + "<title>" + "bonjour" + "</title>"
           + "<body><h1>" + "bonjour" + "</h1></body>" + "</html> ";
    }
}
```

Creating the `web.xml` configuration file

- The Web XML Configuration file which is an XML file to specify the Jersey framework servlet for the application.
- Create a new file under the `src/main/webapp/WEB-INF` folder named `web.xml` in Eclipse with the following text

Creating the `web.xml` configuration file

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >
  <web-app>
    <display-name>Hello</display-name>
     <servlet>
        <servlet-name>jersey-servlet</servlet-name>
        <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
        <init-param>
             <param-name>jersey.config.server.provider.packages</param-name>
             <param-value>examples.hello</param-value>
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>
       <servlet-mapping>
        <servlet-name>jersey-servlet</servlet-name>
        <url-pattern>/rest/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Creating the `web.xml` configuration file

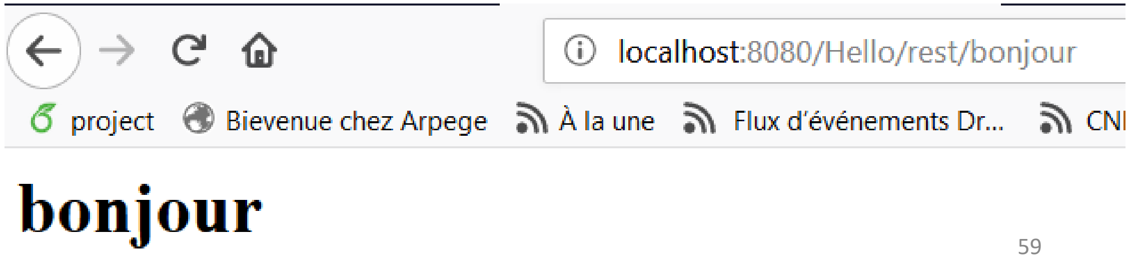
- This file describes the servlet
  - `<servlet-class>` is the name of the Jersey servlet container
  - `<param-value>` is the package name of the service POJOs
  - `<url-pattern>` specifies the URI patterns that can be served
  - `<servlet-name>` must be the same in the `<servlet>` and in the `<servlet-mapping>` sections

Deploying the program

- Once we are done with creating the source and web configuration files, we need to export the application as a **war** file and deploy the same in Tomcat.
- Do **File → export → Web → War File** and finally select the project Hello and the destination folder.
- This destination folder is
    - Tomcat Installation Directory → **webapps directory (for Tomcat 9)**
    - Tomcat Installation Directory → **webapps-javaee directory (for Tomcat 10)**
- Start Tomcat

Understanding how it works

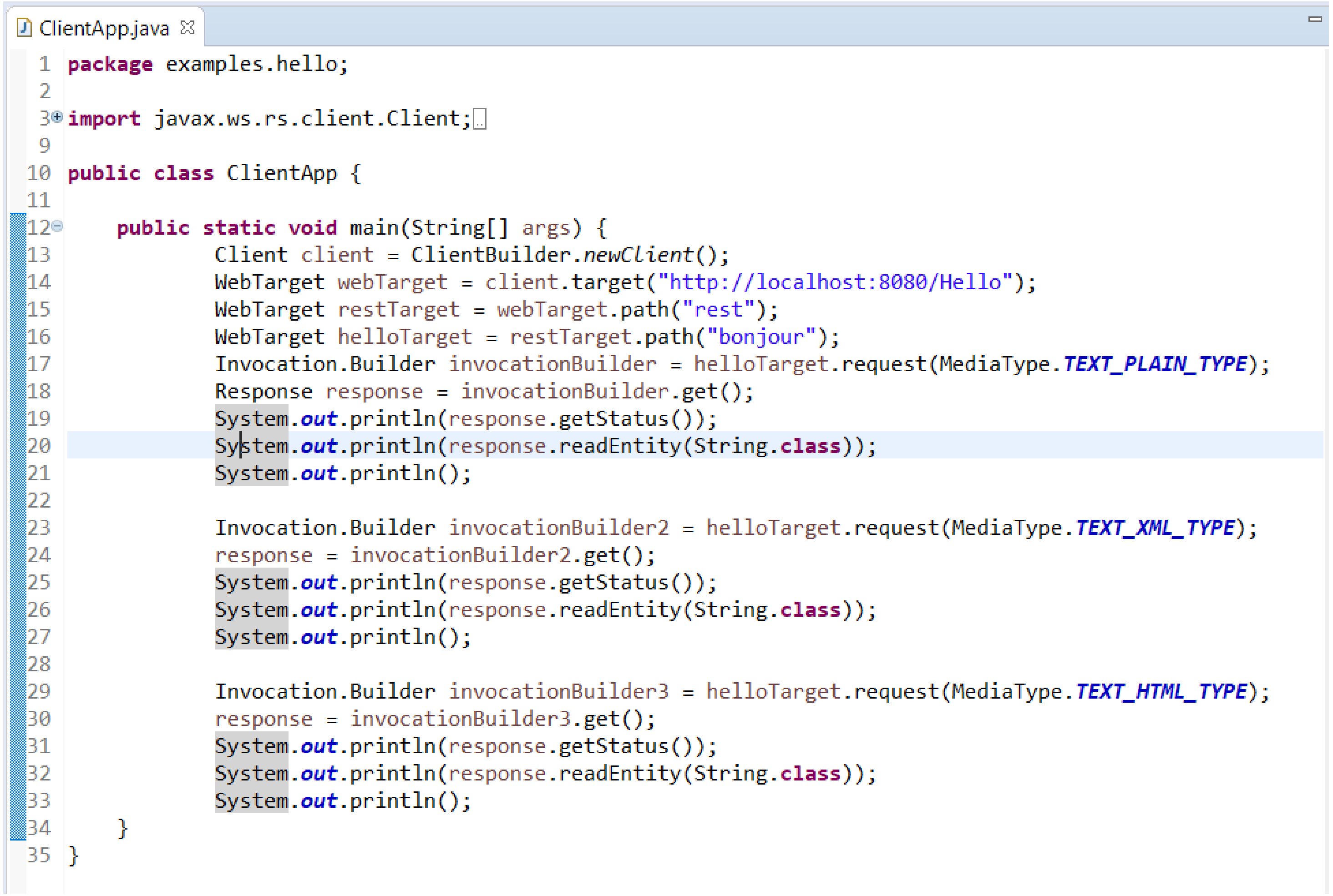
- The project is called Hello
    - It is the base of the path to the REST resource
- Doing http://localhost:8080/Hello/rest/bonjour
    - Launches the `Hello` servlet using Jersey
    - As the URI is `/rest/*` , in the package indicated in **web.xml**, a method accepting a GET request is looked for, with bonjour in its path
    - If you test the URI with a browser, implicitly the GET request producing an HTML output is chosen



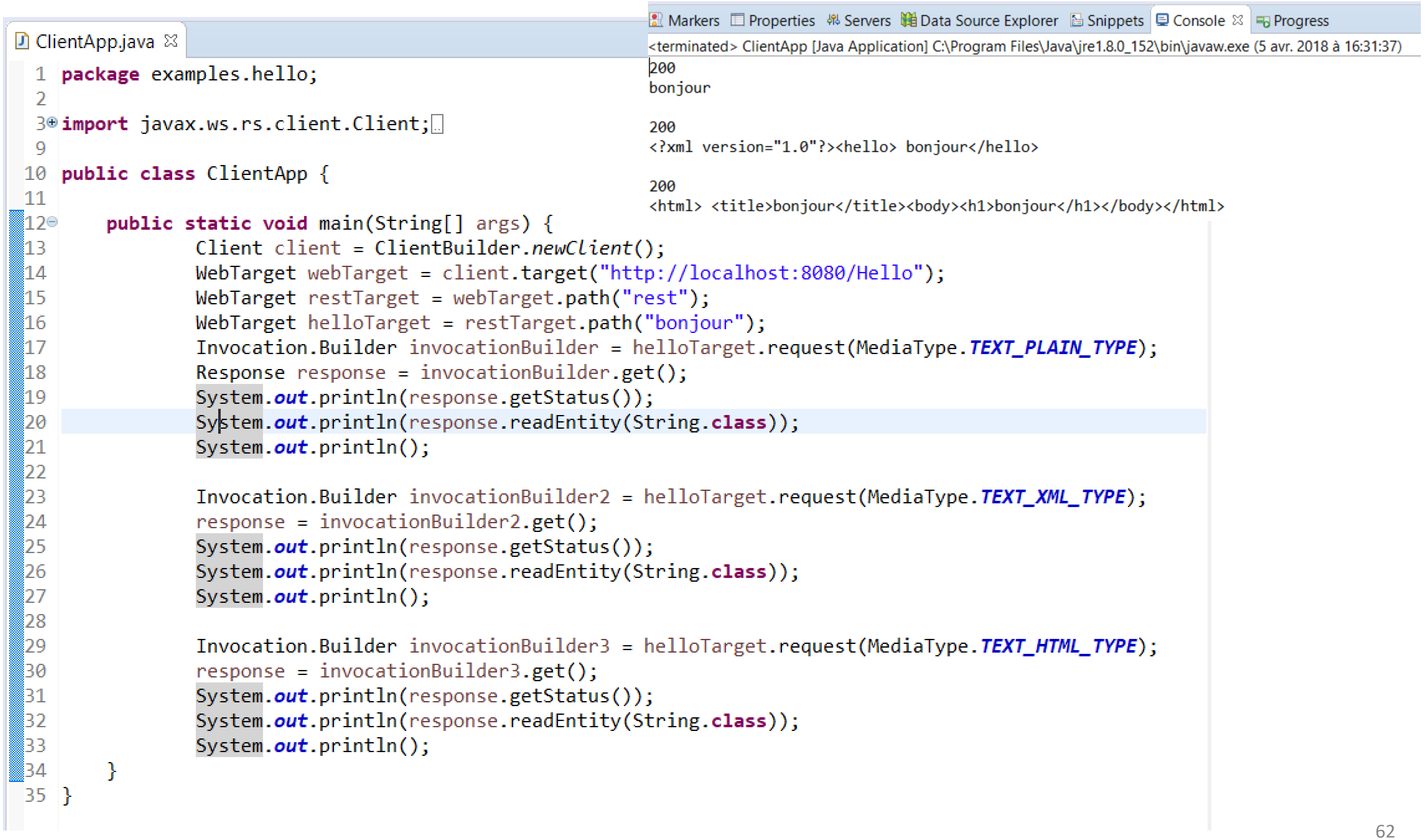
bonjour

Testing the application with a client

- JAX-RS provides elements to write a Client application to consume a REST service
- In the same `examples.hello` package create a new class `ClientApp.java`
  - You can download it
- Run it as a Java Application and analyse the results


ClientApp.java

```java
package examples.hello;

import javax.ws.rs.client.Client;

public class ClientApp {

    public static void main(String[] args) {
            Client client = ClientBuilder.newClient();
            WebTarget webTarget = client.target("http://localhost:8080/Hello");
            WebTarget restTarget = webTarget.path("rest");
            WebTarget helloTarget = restTarget.path("bonjour");
            Invocation.Builder invocationBuilder = helloTarget.request(MediaType.TEXT_PLAIN_TYPE);
            Response response = invocationBuilder.get();
            System.out.println(response.getStatus());
            System.out.println(response.readEntity(String.class));
            System.out.println();

            Invocation.Builder invocationBuilder2 = helloTarget.request(MediaType.TEXT_XML_TYPE);
            response = invocationBuilder2.get();
            System.out.println(response.getStatus());
            System.out.println(response.readEntity(String.class));
            System.out.println();

            Invocation.Builder invocationBuilder3 = helloTarget.request(MediaType.TEXT_HTML_TYPE);
            response = invocationBuilder3.get();
            System.out.println(response.getStatus());
            System.out.println(response.readEntity(String.class));
            System.out.println();
    }
}
```



```
<terminated> ClientApp [Java Application] C:\Program Files\Java\jre1.8.0_152\bin\javaw.exe (5 avr. 2018 à 16:31:37)
200
bonjour

200
<?xml version="1.0"?><hello> bonjour</hello>

200
<html> <title>bonjour</title><body><h1>bonjour</h1></body></html>
```

```java
package examples.hello;

import javax.ws.rs.client.Client;

public class ClientApp {

    public static void main(String[] args) {
            Client client = ClientBuilder.newClient();
            WebTarget webTarget = client.target("http://localhost:8080/Hello");
            WebTarget restTarget = webTarget.path("rest");
            WebTarget helloTarget = restTarget.path("bonjour");
            Invocation.Builder invocationBuilder = helloTarget.request(MediaType.TEXT_PLAIN_TYPE);
            Response response = invocationBuilder.get();
            System.out.println(response.getStatus());
            System.out.println(response.readEntity(String.class));
            System.out.println();

            Invocation.Builder invocationBuilder2 = helloTarget.request(MediaType.TEXT_XML_TYPE);
            response = invocationBuilder2.get();
            System.out.println(response.getStatus());
            System.out.println(response.readEntity(String.class));
            System.out.println();

            Invocation.Builder invocationBuilder3 = helloTarget.request(MediaType.TEXT_HTML_TYPE);
            response = invocationBuilder3.get();
            System.out.println(response.getStatus());
            System.out.println(response.readEntity(String.class));
            System.out.println();
    }
}
```

How to build the ClientApps

- Create an Instance of a `Client`
- Once you have the `Client` instance, you can create a `WebTarget` using the URI of the targeted web resource.
  - Using *`WebTarget`*, you can define a path to a specific resource
- Build an HTTP Request Invocation
  - An `invocation builder` instance is created with one of the `WebTarget.request()` methods
  - Analyse the different formats.
  - You can also use `MediaType.APPLICATION.JSON`

How to build the ClientApps

- Invoke HTTP Requests
  - HTTP GET:
    ```
    Response response  =
                              invocationBuilder.get(XXX.class);
    ```
  - HTTP POST:
    ```
    Response response
        =invocationBuilder.post
       (Entity.entity(XXX,MediaType.APPLICATION_JSON);
    ```

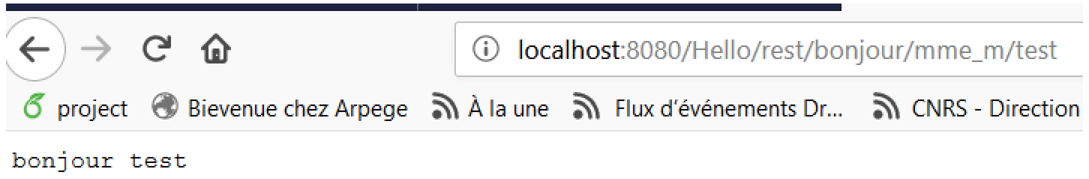
- *XXX is generally the name of the class whose objects are to get or to send*

Passing parameters in the GET query

- At the server side, add

```
@Path("/mme_m/{nom}")
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello2(@PathParam("nom") String nom) {
     return "bonjour " + nom;
}
```

- Test on a browser



localhost:8080/Hello/rest/bonjour/mme_m/test

```
bonjour test
```

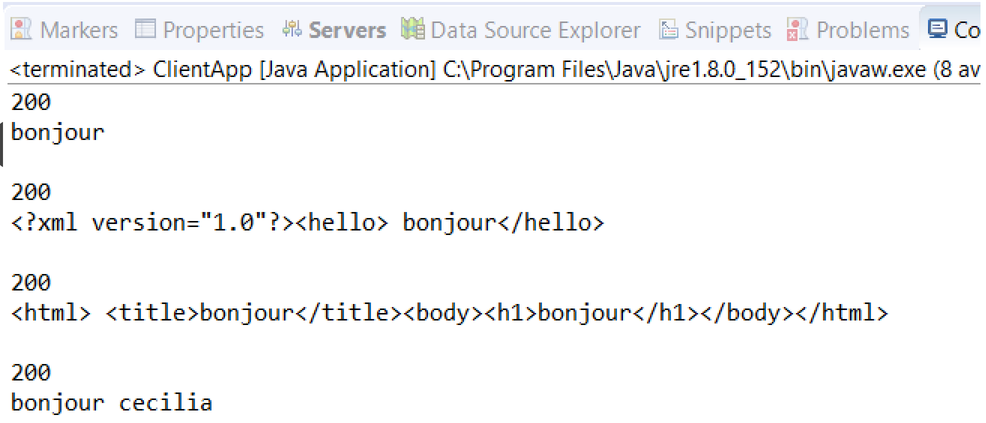
Passing parameters in the GET query

- At the client side, add

```
WebTarget helloTargetParam =
webTarget.path("rest").path("bonjour").path("mme_m").path("cecilia");

Invocation.Builder invocationBuilder4 =
helloTargetParam.request(MediaType.TEXT_PLAIN_TYPE);

response = invocationBuilder4.get();

System.out.println(response.getStatus());

System.out.println(response.readEntity(String.class));

System.out.println();
```

Passing parameters in the

- At the client side, add



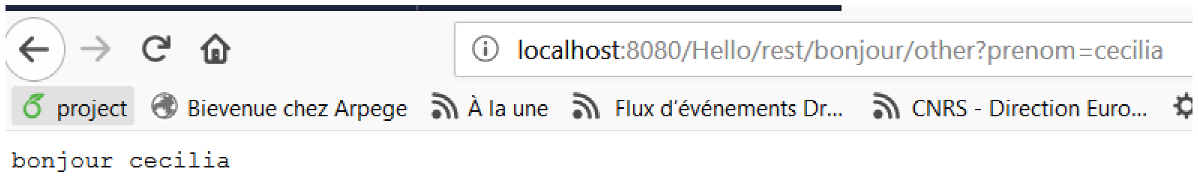
```
<terminated> ClientApp [Java Application] C:\Program Files\Java\jre1.8.0_152\bin\javaw.exe (8 av
200
bonjour

200
<?xml version="1.0"?><hello> bonjour</hello>

200
<html> <title>bonjour</title><body><h1>bonjour</h1></body></html>

200
bonjour cecilia
```

```
WebTarget helloTargetParam =
webTarget.path("rest").path("bonjour").path("mme_m").path("cecilia");
Invocation.Builder invocationBuilder4 =
helloTargetParam.request(MediaType.TEXT_PLAIN_TYPE);
response = invocationBuilder4.get();
System.out.println(response.getStatus());
System.out.println(response.readEntity(String.class));
System.out.println();
```

Passing parameters in the `?param=value` part

- At the server side, add

```
@Path("/other")
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello3(@DefaultValue("inconnu")
                     @QueryParam("prenom") String pre) {
     return "bonjour " + pre;
}
```



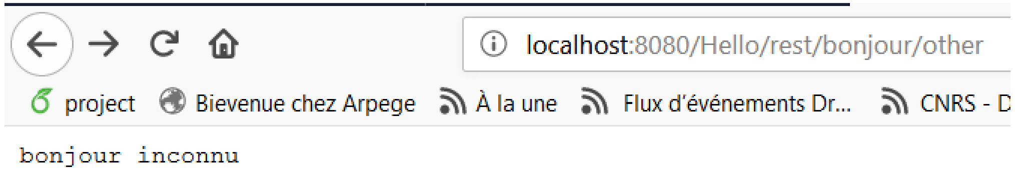
```
localhost:8080/Hello/rest/bonjour/other?prenom=cecilia
```

```
bonjour cecilia
```

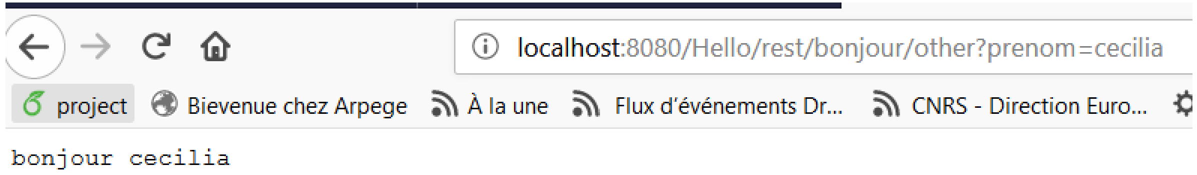
Passing parameters in the `?param=value` part

- At the server side, add

```
@Path("/other")
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello3(@DefaultValue("inconnu")
                                 @QueryParam("prenom") String
prenom) {
     return "bonjour " + prenom;
}
```



```
bonjour inconnu
```



```
bonjour cecilia
```

Passing parameters in the `?param=value` part

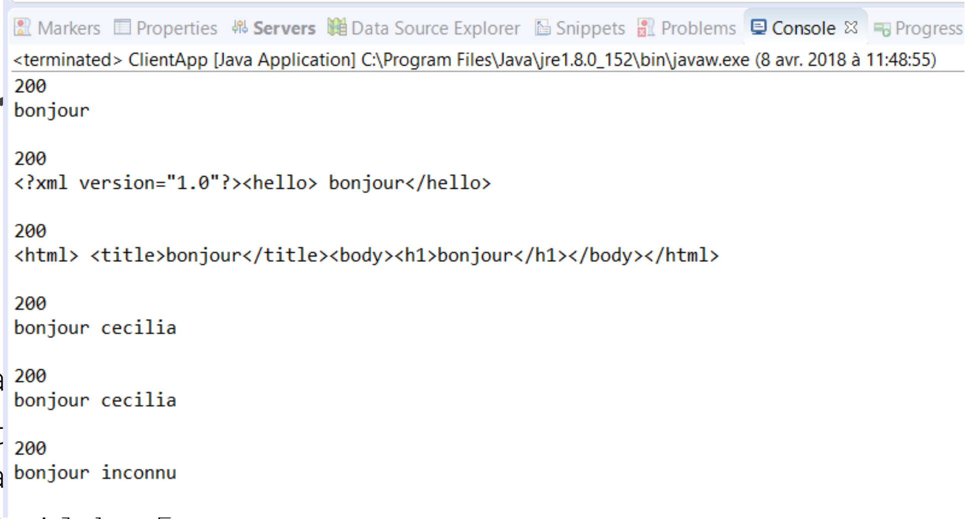
- At the client side, add

```
WebTarget helloTargetParam2 =
     webTarget.path("rest").path("bonjour").path("other");
WebTarget helloWithparamTarget =
     helloTargetParam2.queryParam("prenom", "cecilia");
Invocation.Builder invocationBuilder5 =
     helloWithparamTarget.request(MediaType.TEXT_PLAIN_TYPE);
response = invocationBuilder5.get();
…
Invocation.Builder invocationBuilder6 =
     helloTargetParam2.request(MediaType.TEXT_PLAIN_TYPE);
response = invocationBuilder6.get();
…
```

Passing parameters in the

- At the client side, add

```
WebTarget helloTargetParam2 =
     webTarget.path("rest").pa
WebTarget helloWithparamTarget
     helloTargetParam2.queryPa
Invocation.Builder invocationBuilder5 =
     helloWithparamTarget.request(MediaType.TEXT_PLAIN_TYPE);
response = invocationBuilder5.get();
…
Invocation.Builder invocationBuilder6 =
     helloTargetParam2.request(MediaType.TEXT_PLAIN_TYPE);
response = invocationBuilder6.get();
…
```


Markers Properties Servers Data Source Explorer Snippets Problems Console Progress

```
<terminated> ClientApp [Java Application] C:\Program Files\Java\jre1.8.0_152\bin\javaw.exe (8 avr. 2018 à 11:48:55)
200
bonjour

200
<?xml version="1.0"?><hello> bonjour</hello>

200
<html> <title>bonjour</title><body><h1>bonjour</h1></body></html>

200
bonjour cecilia

200
bonjour cecilia

200
bonjour inconnu
```

Summarizing: `@Path`

```
@Path("/bonjour")
public class Hello {

@Path("/other")
@GET
public String sayPlainTextHello3( …
…
@Path("/mme_m/{nom}")
@GET
public String sayPlainTextHello2 ...
```

Summarizing: `@PathParam`

```
@Path("/mme_m/{nom}")
@GET
public String          sayPlainTextHello2(@PathParam("nom")
                       String nom) {
...
```

Attention:
The types that can be passed as parameters are
1. Primitive types
2. String
3. Type/Class having a constructor with an only argument of type String
4. Type/Class having a static method ValueOf(String)
5. Among other more complex combinations .. https://docs.oracle.com/javaee/7/api/javax/ws/rs/PathParam.html

Summarizing: `@QueryParam @DefaultValue`

```
@Path("/other")
@GET
public StringsayPlainTextHello3(@DefaultValue("inconnu")
                    @QueryParam("prenom") String prenom)
{
```

Summarizing: @Produces @Consumes

```
@GET
@Produces(MediaType.TEXT_PLAIN)
public String sayPlainTextHello() {
...
```

Summarizing: `@GET @POST @PUT @DELETE`

```
@GET
public String sayPlainTextHello() {
...
```

Up to you!!!

- Redo the exercise done with SOAP web services.
  - Develop a REST web service to calculate the body mass index of a client. The body mass index is calculated as the ratio between the weight of the person (in kg) and the square of his or her height (in m).
  - Your server, therefore, should have a method that receives two parameters.

Passing objects as parameters

- JAX-RS allows to pass objects of type other than those mentioned above, by relying on the marshalling/unmarshalling provided by JAXB (Java Architecture for XML Binding): serialization of a JAVA object into an XML document and inversely
- JAX-RS is also able to serialize/deserialize in JSON.
- JAXB provides annotations that, when applied to a POJO, simplify the transformation.
  - `@XmlRootElement` defines the root of the XML document generated from this class.

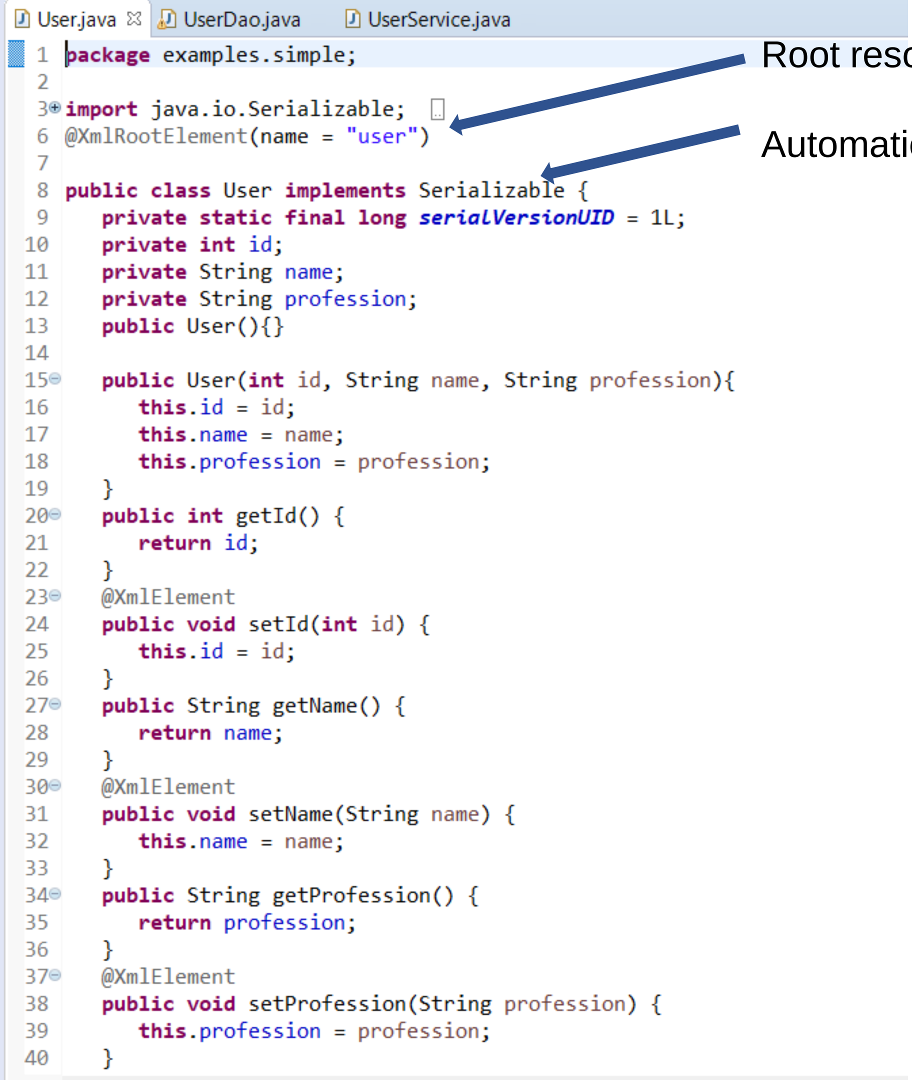
An more complex example

- Let us create a web service called **User Management** with the following functionalities

Service	HTTP Method	URI	Operation	Operation Type
1	GET	/UserService/users	Get list of users	Read Only
2	GET	/UserService/users/1	Get User with Id 1	Read Only
3	PUT	/UserService/users/2	Update User with Id 2	Idempotent
4	POST	/UserService/users/2	Create User with Id 2	N/A
5	DELETE	/UserService/users/1	Delete User with Id 1	Idempotent

Implementing service 1

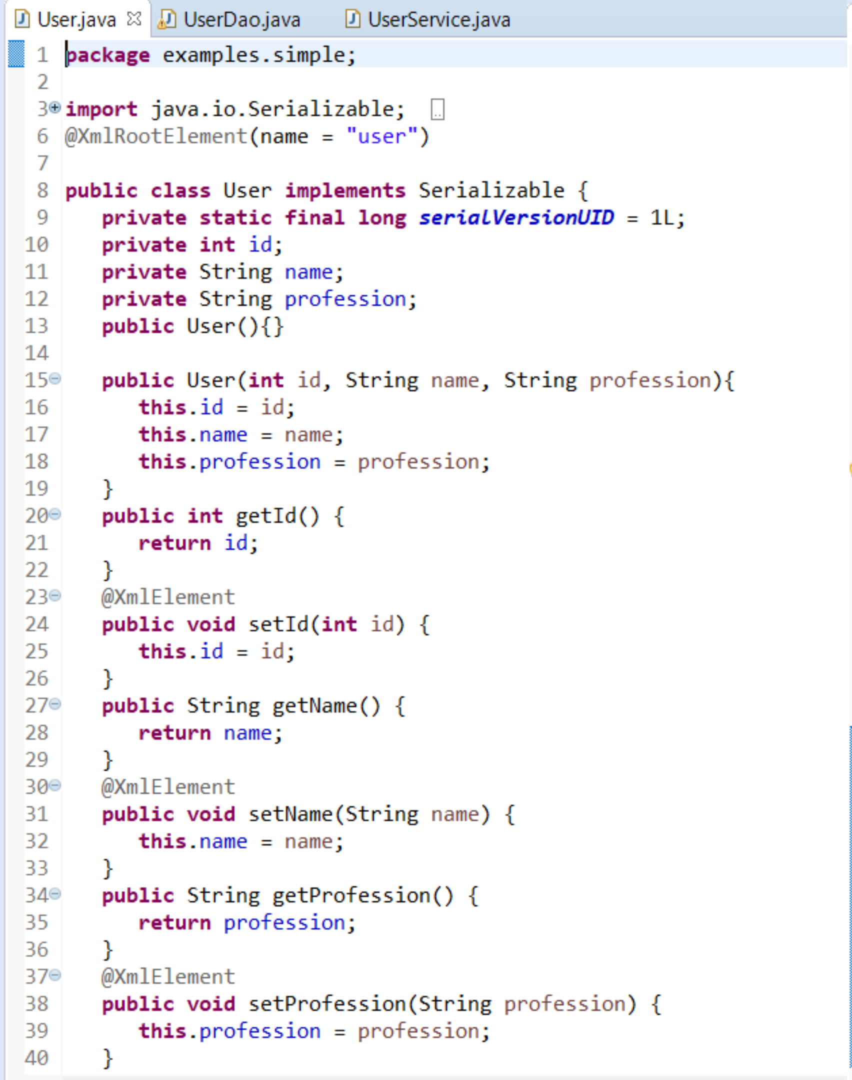
- Organize a new Eclipse project, with a package named `examples.simple`
- You will need these 3 classes (that you can download)
  - `User.java` ← This is the **root resource class**, as already discussed (slide 43)
  - `UserDao.java` ←This class manages the file where the users will be stored
  - `UserService.java` ← This is the “real” server class
- Remember to edit the `web.xml` file to update the package name



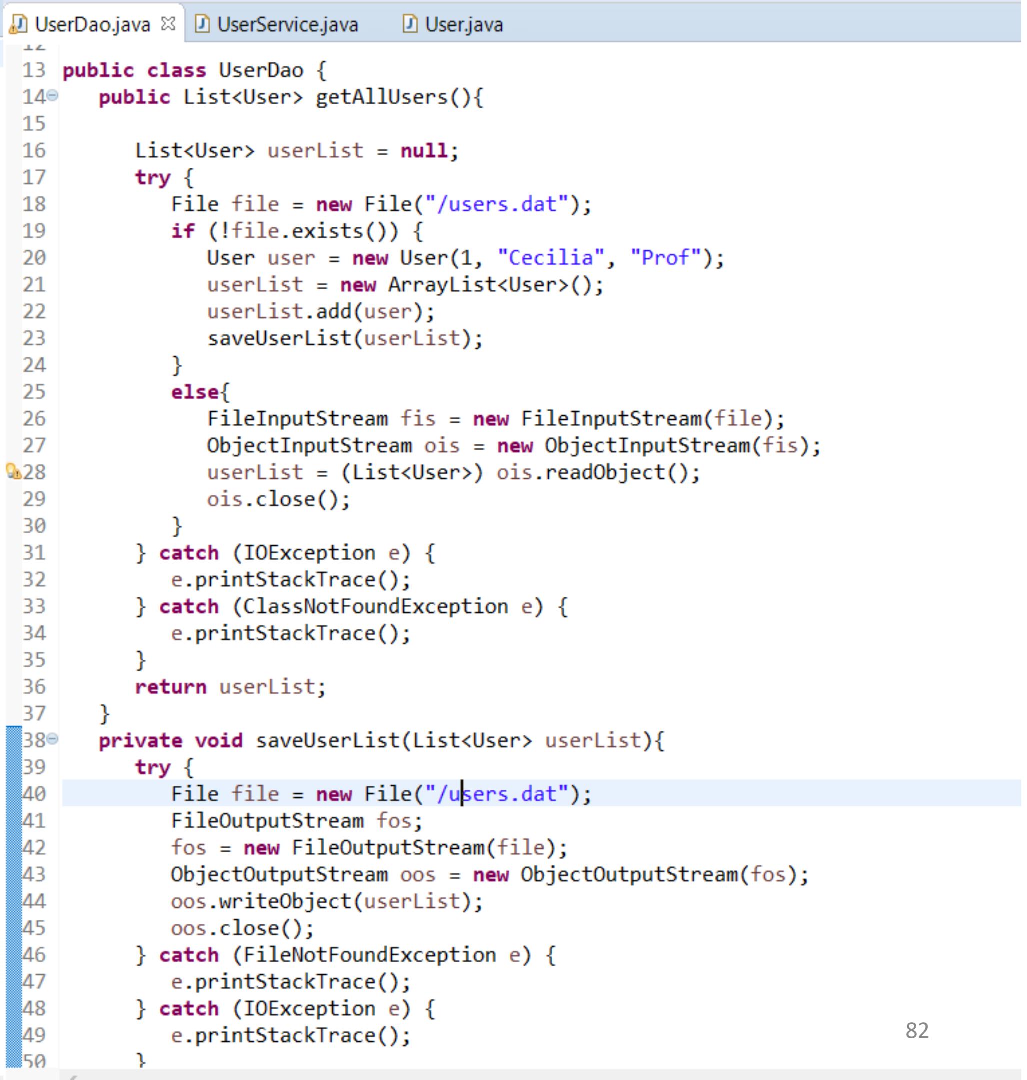
```
package examples.simple;

import java.io.Serializable;
@XmlRootElement(name = "user")

public class User implements Serializable {
   private static final long serialVersionUID = 1L;
   private int id;
   private String name;
   private String profession;
   public User(){}

   public User(int id, String name, String profession){
      this.id = id;
      this.name = name;
      this.profession = profession;
   }
   public int getId() {
      return id;
   }
   @XmlElement
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   @XmlElement
   public void setName(String name) {
      this.name = name;
   }
   public String getProfession() {
      return profession;
   }
   @XmlElement
   public void setProfession(String profession) {
      this.profession = profession;
   }
```

Root resource class

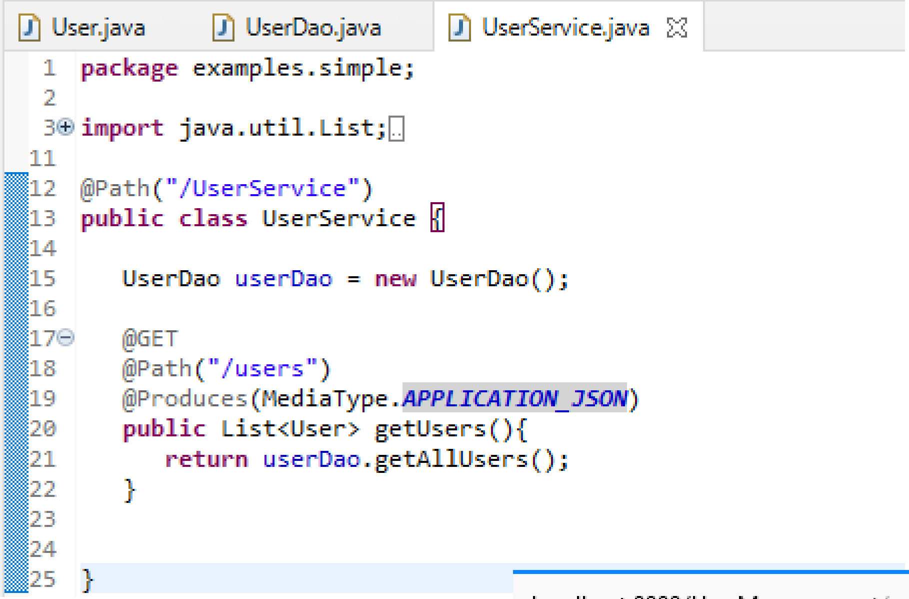
Automatic serialization to use structured objects


User.java ☒ | UserDao.java | UserService.java

```java
package examples.simple;

import java.io.Serializable;
@XmlRootElement(name = "user")

public class User implements Serializable {
   private static final long serialVersionUID = 1L;
   private int id;
   private String name;
   private String profession;
   public User(){}

   public User(int id, String name, String profession){
      this.id = id;
      this.name = name;
      this.profession = profession;
   }
   public int getId() {
      return id;
   }
   @XmlElement
   public void setId(int id) {
      this.id = id;
   }
   public String getName() {
      return name;
   }
   @XmlElement
   public void setName(String name) {
      this.name = name;
   }
   public String getProfession() {
      return profession;
   }
   @XmlElement
   public void setProfession(String profession) {
      this.profession = profession;
   }
```

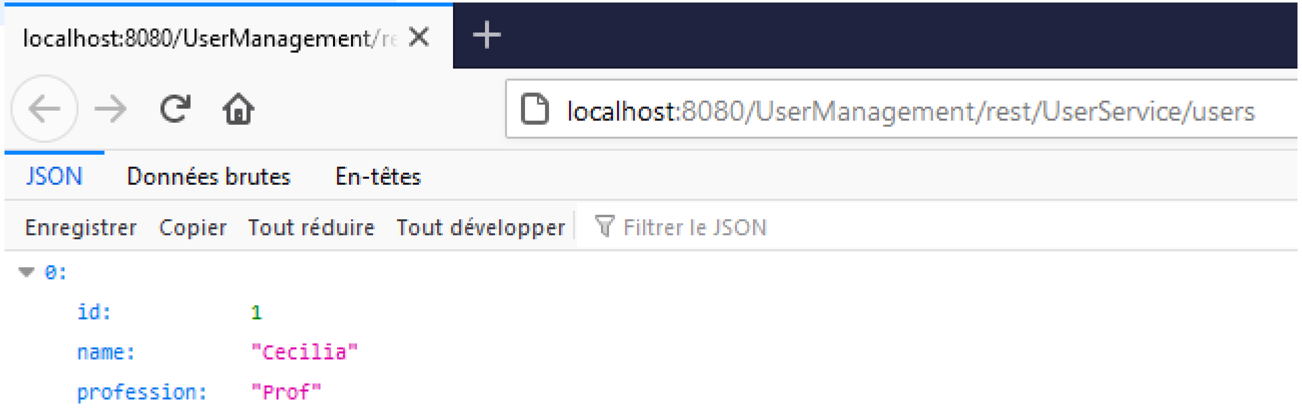

UserDao.java ☒ | UserService.java | User.java

```java
public class UserDao {
   public List<User> getAllUsers(){

      List<User> userList = null;
      try {
         File file = new File("/users.dat");
         if (!file.exists()) {
            User user = new User(1, "Cecilia", "Prof");
            userList = new ArrayList<User>();
            userList.add(user);
            saveUserList(userList);
         }
         else{
            FileInputStream fis = new FileInputStream(file);
            ObjectInputStream ois = new ObjectInputStream(fis);
            userList = (List<User>) ois.readObject();
            ois.close();
         }
      } catch (IOException e) {
         e.printStackTrace();
      } catch (ClassNotFoundException e) {
         e.printStackTrace();
      }
      return userList;
   }
   private void saveUserList(List<User> userList){
      try {
         File file = new File("/users.dat");
         FileOutputStream fos;
         fos = new FileOutputStream(file);
         ObjectOutputStream oos = new ObjectOutputStream(fos);
         oos.writeObject(userList);
         oos.close();
      } catch (FileNotFoundException e) {
         e.printStackTrace();
      } catch (IOException e) {
         e.printStackTrace();
      }
```



User.java UserDao.java UserService.java

```java
package examples.simple;

import java.util.List;

@Path("/UserService")
public class UserService {

   UserDao userDao = new UserDao();

   @GET
   @Path("/users")
   @Produces(MediaType.APPLICATION_JSON)
   public List<User> getUsers(){
      return userDao.getAllUsers();
   }


}
```



localhost:8080/UserManagement/rest/UserService/users

```
▼ 0:
    id:          1
    name:        "Cecilia"
    profession:  "Prof"
```

Up to you!!!

- Create a client to consume this service 1
- Update the server to implement service 2 (get the user with ID 1 in `users.dat`)
- Create a client to consume service 2

Implementing services 4 and 5

- You will need to update `UserDao.java` and `UserService.java`
- In `UserDao.java`, add methods to add and to delete a user.

Implementing services 4 and 5

- You will need to update `UserDao.java` and `UserService.java`
- In `UserDao.java`, add methods to add and to delete a user.

```
public int addUser(User pUser){
      List<User> userList = getAllUsers();
      boolean userExists = false;
      for(User user: userList){
         if(user.getId() == pUser.getId()){
            userExists = true;
            break;
         }
      }
      if(!userExists){
         userList.add(pUser);
         saveUserList(userList);
         return 1;
      }
      return 0;
   }
```

```
public int deleteUser(int id){
      List<User> userList = getAllUsers();

      for(User user: userList){
         if(user.getId() == id){
            int index = userList.indexOf(user);
            userList.remove(index);
            saveUserList(userList);
            return 1;
         }
      }
      return 0;
   }
```

Implementing services 4 and 5

- In `UserService.java`, you need to add the services. First of all, after the initialisation of the `UserDao`, add 2 constants

```
private static final String
SUCCESS_RESULT="<result>success</result>";
private static final String
FAILURE_RESULT="<result>failure</result>";
```

Implementing services 4 and 5

```
 @POST
   @Path("/users")
   @Produces(MediaType.APPLICATION_XML)

@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
   public String createUser(@FormParam("id") int
id,
      @FormParam("name") String name,
      @FormParam("profession") String
profession,
      @Context HttpServletResponse
servletResponse) throws IOException{
      User user = new User(id, name,
profession);
      int result = userDao.addUser(user);
      if(result == 1){
         return SUCCESS_RESULT;
      }
      return FAILURE_RESULT;
   }
```

```
 @DELETE
   @Path("/users/{userid}")
   @Produces(MediaType.APPLICATION_XML)
   public String
deleteUser(@PathParam("userid") int
userid){
      int result =
userDao.deleteUser(userid);
      if(result == 1){
         return SUCCESS_RESULT;
      }
      return FAILURE_RESULT;
   }
```

Implementing services 4 and 5

```
 @POST
   @Path("/users")
   @Produces(MediaType.APPLICATION_XML)

@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
   public String createUser(@FormParam("id") int
id,
      @FormParam("name") String name,
      @FormParam("profession") String
profession,
      @Context HttpServletResponse
servletResponse) throws IOException{
      User user = new User(id, name,
profession);
      int result = userDao.addUser(user);
      if(result == 1){
         return SUCCESS_RESULT;
      }
      return FAILURE_RESULT;
   }
```

```
 @DELETE
   @Path("/users/{userid}")
   @Produces(MediaType.APPLICATION_XML)
   public String
deleteUser(@PathParam("userid") int
userid){
      int result =
userDao.deleteUser(userid);
      if(result == 1){
         return SUCCESS_RESULT;
      }
      return FAILURE_RESULT;
   }
```

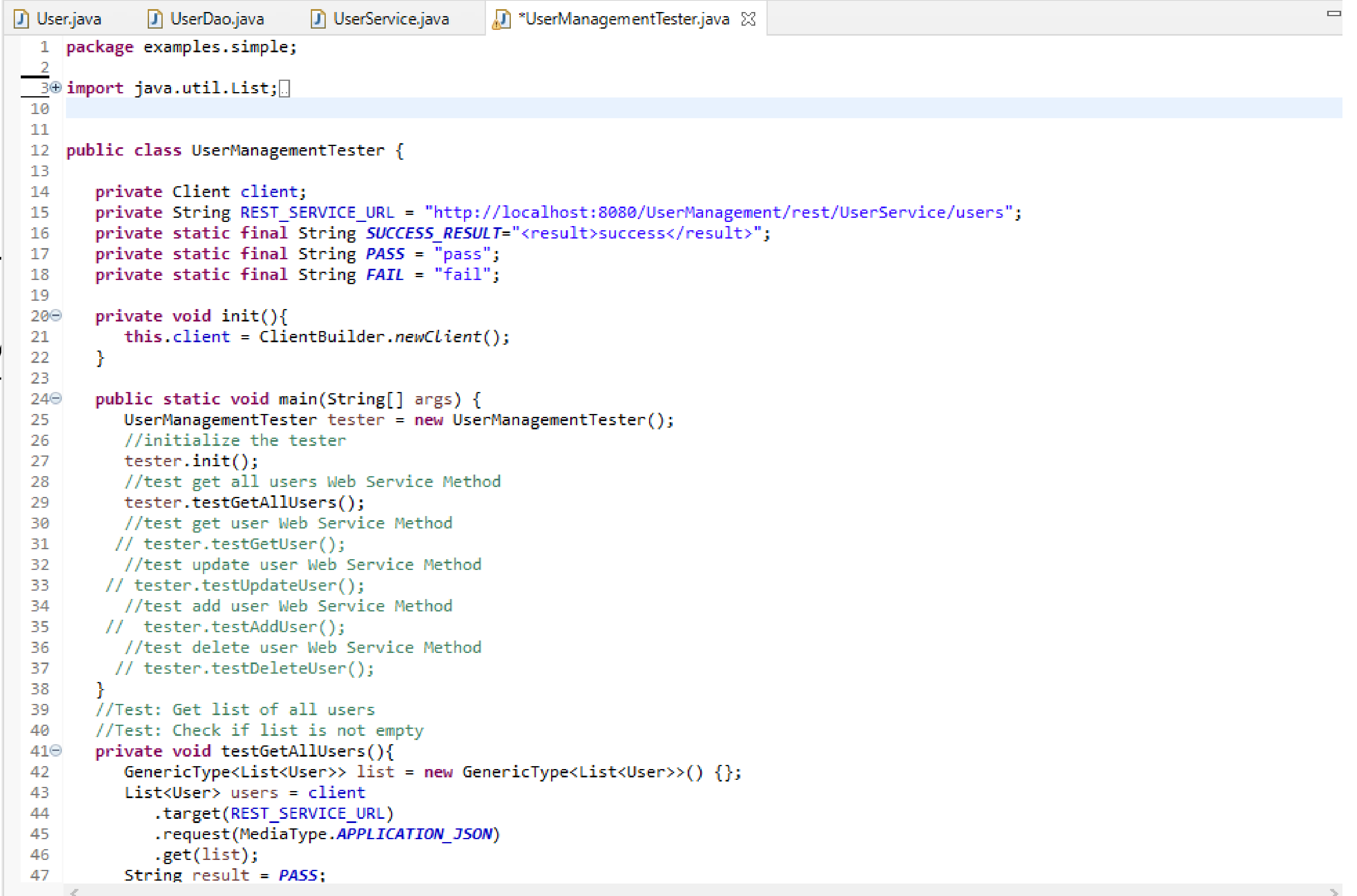
Remark:
`MediaType.APPLICATION_FORM_URLENCODED`
The parameters will be passed through 'Form' values as key-value pair. 'Key' should match with the @FormParam annotation value.

Testing …

- Download and test `UserManagementTester.java`
    - All the code necessary to test all the services is already there. Analyse it carefully.

Testi

- Dowr
  - /



```java
package examples.simple;

import java.util.List;


public class UserManagementTester {

   private Client client;
   private String REST_SERVICE_URL = "http://localhost:8080/UserManagement/rest/UserService/users";
   private static final String SUCCESS_RESULT="<result>success</result>";
   private static final String PASS = "pass";
   private static final String FAIL = "fail";

   private void init(){
      this.client = ClientBuilder.newClient();
   }

   public static void main(String[] args) {
      UserManagementTester tester = new UserManagementTester();
      //initialize the tester
      tester.init();
      //test get all users Web Service Method
      tester.testGetAllUsers();
      //test get user Web Service Method
     // tester.testGetUser();
      //test update user Web Service Method
    // tester.testUpdateUser();
      //test add user Web Service Method
    //  tester.testAddUser();
      //test delete user Web Service Method
     // tester.testDeleteUser();
   }
   //Test: Get list of all users
   //Test: Check if list is not empty
   private void testGetAllUsers(){
      GenericType<List<User>> list = new GenericType<List<User>>() {};
      List<User> users = client
         .target(REST_SERVICE_URL)
         .request(MediaType.APPLICATION_JSON)
         .get(list);
      String result = PASS;
```

Up to you!!!

- Implement the PUT service (service number 3)
- Modify the `UserManagementTester.java` class to test the new service you have implemented
- The POST service we have implemented expects arguments as `FORM-URLENCODED`. Develop another service for the POST method, where the arguments are in XML format, so that you can test it with the **RESTClient** plug-in for Firefox
  - Test it!!!

Some Final Remarks

- The implementation of RESTful services avoids the creation of many Java servlets for ordinary web form actions
- Implementation in JAX-RS with Jersey is almost transparent
- However, care must be taken to properly manage the idempotence of requests in order to avoid duplicates or copies.