

Web Services

Daniel Hagimont

<https://www.google.fr/search?q=daniel+hagimont+home+page>

1

This lecture is about web services.

Motivations

- Motivations
 - Coarse-grained application integration
 - Unit of integration: the "service" (interface + contract)
- Constraints
 - Applications developed independently, without anticipation of any integration
 - Heterogeneous applications (models, platforms, languages)
- Consequences
 - Elementary common basis
 - For communication protocols (messages)
 - For the description of services (interface)
 - Base choice: HTTP and XML/JSON

2

The example of RPC tool we have seen, Java RMI, is restricted to interactions within Java applications, allowing remote invocations of Java objects.

With Web services, the motivation is to provide a RPC facility for the interaction (and integration) of coarse-grained applications (that we call services). A service is supposed to be much bigger than a simple Java object.

Web services aim at allowing the interaction between application developed independently, with different environments (models, platforms, languages).

Web services rely on elementary existing protocols and formats: mainly HTTP, XML and JSON.

Basic form of WS : XML-RPC (1998)

Description in XML of a remote procedure call
Parameter types are specified in an XML schema

```
<methodCall>
  <methodName>meteo.temperature</methodName>
  <params>
    <param>
      <value><int>31130</int></value>
    </param>
  </params>
</methodCall>
```

Description in XML of parameter returns

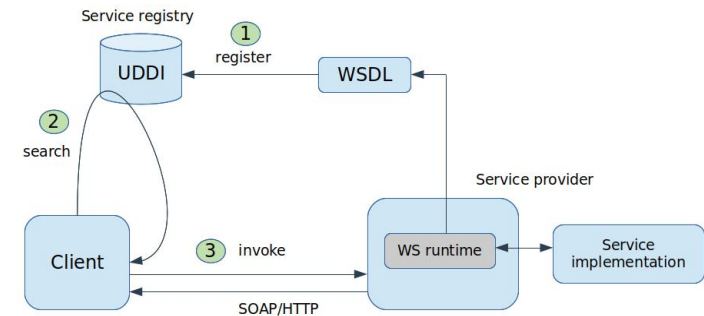
```
<methodResponse>
  <params>
    <param>
      <value><int>25</int></value>
    </param>
  </params>
</methodResponse>
```

Interest : independence with respect to
platforms and communication protocols

<https://en.wikipedia.org/wiki/XML-RPC>

3

SOAP WS : architecture (2000++)



4

This figure illustrates the architecture of web services (WS).

A service provider may implement a service in any language and/or platform, as soon as a runtime for WS exists in his environment.

The runtime is a composed of

- stub and skeleton generators
- a WSDL generator
- a web server (WS runtime) for making services available on the internet

Then, on the server side, the service implementation is linked with the web server, in order to be able to receive requests through the HTTP communication protocol. A skeleton is generated and is a web application in the web server. A WSDL description (Web Service Description Language) of the service is generated and published, i.e. made available to potential clients.

The WS architecture specifies that a service registry (a naming service) should be used for the publication and discovery of WSDL descriptions. However, UDDI was not actually used. Generally the WSDL of a WS can be published on a Web server as any document.

On the client side, the WSDL description can be copied and used to generate a stub in the environment of the client. Notice that the environment of the client is not mandatorily the same as the one of the server. Then the client can implement an application which is able to invoke the WS by calling the stub.

The stub communicates with the skeleton with the SOAP/HTTP protocol which is a standard.

HTTP and SOAP are standards from the W3C.

SOAP describes the syntax of request and response messages which are transported with HTTP.

Elements of SOAP WS

- Description of a service
 - WSDL : Web Services Description Language
 - Standard notation for the description of a service interface
- Access to a service
 - SOAP : Simple Object Access Protocol (over HTTP)
 - Internet protocol allowing communication between Web Services
- Registry of services
 - UDDI : Universal Description, Discovery and Integration
 - Protocol for registration and discovery of services

5

Tools

- From a program, we can generate a WS skeleton
 - Example: from a Java program, we generate
 - A servlet which receives SOAP/HTTP requests and reproduces the invocation on an instance of the class
 - A WDSL file which describes the WS interface
- The generated WSDL file can be given to clients
- From WSDL file, we can generate a WS stub
 - Example: from a WSDL file, we generate Java classes which can be used to invoke the remote service
- Programming is simplified
- Such tools are available in different language environments

6

Therefore the main elements of WS are :

- the description of the service in WSDL. Generally, from an implementation of a service (e.g. a procedure), tools are provided to generate the WSDL description of the service, which is published for clients. The clients can use this WSDL description to generate stubs so that calls to the service can be programmed easily.
- access protocols which are SOAP (for the content of messages) and HTTP (for the transport). All the WS runtimes (in any environment) comply with these standards.
- registries of service (UDDI) which are not really used.

To illustrate this, we give an example of use in the Java environment.

In the Java environment, a WS tool is used to generate from a program (with an exported interface) a skeleton as a servlet. A servlet is a Java program which runs in a web server. This servlet/skeleton receives SOAP/HTTP requests and reproduces the invocation on an instance of the class. The WSDL specification of the WS is also generated.

The WSDL file is published on the web and imported by the client.

From the WSDL specification, the client can generate stubs which make it easier to program WS invocations.

In the following slides, we give an example with Apache Axis.

Example: programming a Web Services

- Eclipse JEE
- Apache Axis
- Creation of a Web Service
 - From a Java class
 - In the Tomcat runtime
 - Generation of the WSDL file
- Creation of a client application
 - Generation of stubs from a WSDL file
 - Programming of the client

7

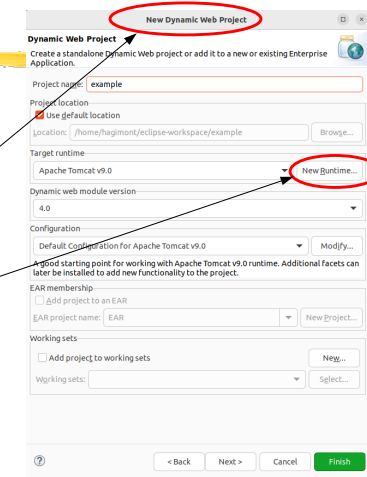
We use Eclipse JEE and Apache Axis which is available in Eclipse JEE.

Apache Axis is used to generate from a Java class a servlet which is installed in the Tomcat engine (the web server). It also generates the WSDL description which describes the interface of the WS.

On the client side, the WSDL description is used to generate stubs which are used to invoke the WS in a client program.

Create a Dynamic Web Project

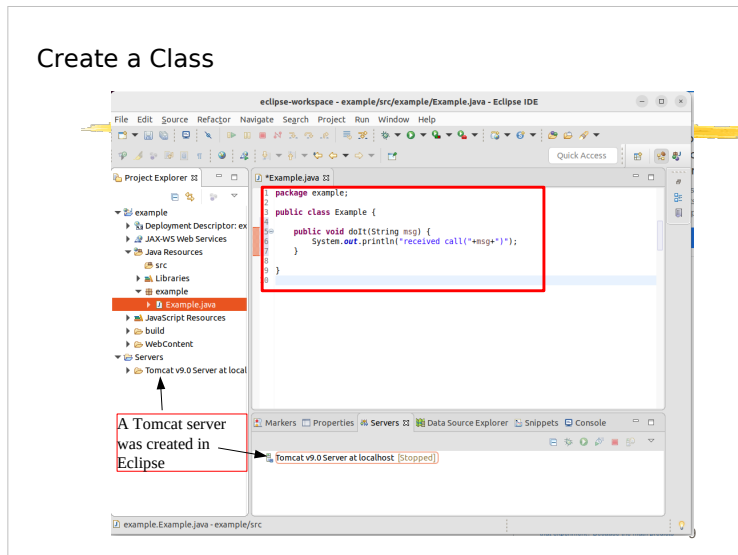
- Eclipse JEE
- Open JEE perspective
- Create a Dynamic Web Project
- Add your Tomcat runtime



8

In Eclipse, we create a dynamic web project (a project allowing the develop servlets) and add the Tomcat runtime.

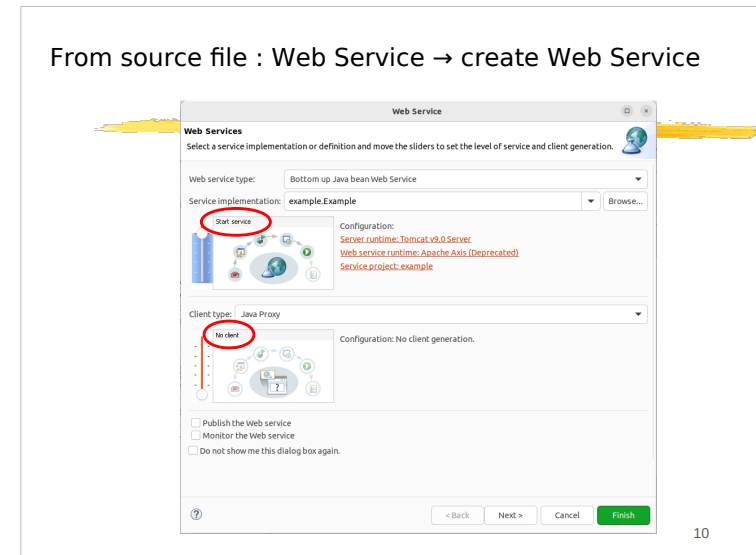
Create a Class



In this project, we create a class.

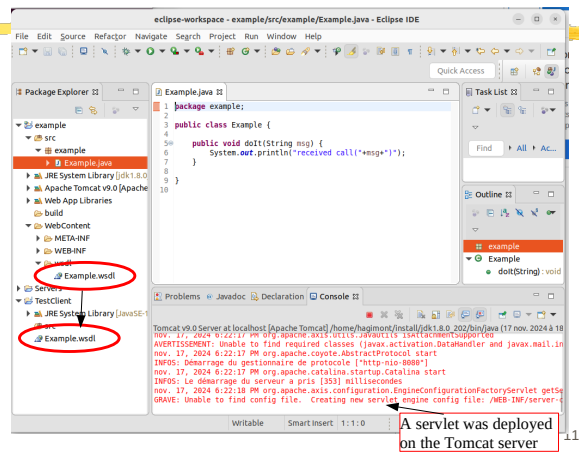
Notice that a Tomcat server is running in Eclipse.

From source file : Web Service → create Web Service

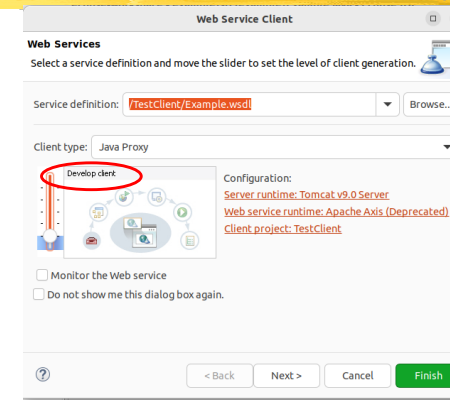


From the source file of the class, we can generate (right click) a WS from this file.

Copy the generated WSDL file in a new Java project



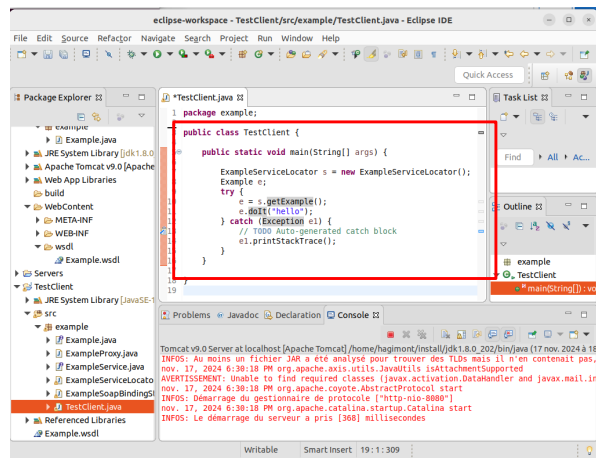
From the WSDL file
Web Service → Generate Client (Develop Client)



Then, we create a new Java project and copy the WSDL description in the new project.

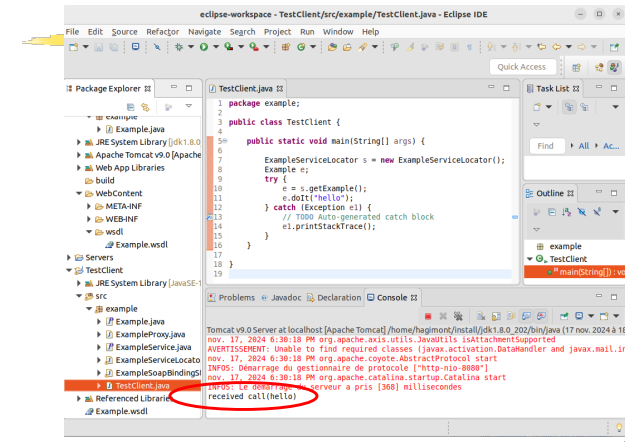
In the new project, from the WSDL file, we generate (right click) the stubs (develop Client).

Program a client



13

Run



14

In the new project, we can program an application which makes an invocation of the WS.

The procedure to follow to invoke the WS depends on the tool used (here Apache Axis).

We can then run the client program which invokes the WS.

Generated WSDL

```
<wsdl:definitions targetNamespace="http://DefaultNamespace"
xmlns:apachesoap="http://xml.apache.org/xml-soap" xmlns:impl="http://DefaultNamespace"
xmlns:intf="http://DefaultNamespace" xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:wsdlsoap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<!--WSDL created by Apache Axis version: 1.4
Built on Apr 22, 2006 (06:55:48 PDT)-->
<wsdl:types>
<schema elementFormDefault="qualified" targetNamespace="http://DefaultNamespace"
xmlns="http://www.w3.org/2001/XMLSchema">
<element name="doIt">
<complexType>
<sequence>
<element name="msg" type="xsd:string"/>
</sequence>
</complexType>
</element>
<element name="doItResponse">
<complexType/>
</element>
</schema>
</wsdl:types>
```

15

We can have a look at the WSDL description.

We can see that the WSDL syntax is not very simple. Therefore such WSDL descriptions are not written by the user, but generally generated by the tool on the server side and imported by the client.

Generated WSDL

```
<wsdl:message name="doItResponse">
<wsdl:part element="impl:doItResponse" name="parameters">
</wsdl:part>
</wsdl:message>
<wsdl:message name="doItRequest">
<wsdl:part element="impl:doIt" name="parameters">
</wsdl:part>
</wsdl:message>
<wsdl:portType name="MyService">
<wsdl:operation name="doIt">
<wsdl:input message="impl:doItRequest" name="doItRequest">
</wsdl:input>
<wsdl:output message="impl:doItResponse" name="doItResponse">
</wsdl:output>
</wsdl:operation>
</wsdl:portType>
```

16

Very verbose !

Generated WSDL

```
<wsdl:binding name="MyServiceSoapBinding" type="impl:MyService">
  <wsdlsoap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <wsdl:operation name="doIt">
    <wsdlsoap:operation soapAction=""/>
    <wsdl:input name="doItRequest">
      <wsdlsoap:body use="literal"/>
    </wsdl:input>
    <wsdl:output name="doItResponse">
      <wsdlsoap:body use="literal"/>
    </wsdl:output>
  </wsdl:operation>
</wsdl:binding>
<wsdl:service name="MyServiceService">
  <wsdl:port binding="impl:MyServiceSoapBinding" name="MyService">
    <wsdlsoap:address location="http://localhost:8080/HW/services/MyService"/>
  </wsdl:port>
</wsdl:service>
</wsdl:definitions>
```

17

Very very verbose !

SOAP request(with TCP/IP Monitor)

```
<?xml version="1.0" encoding="UTF-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <soapenv:Body>
    <doIt xmlns="http://DefaultNamespace">
      <msg>hello</msg>
    </doIt>
  </soapenv:Body>

</soapenv:Envelope>
```

18

We can have a look at the SOAP request. This is simply a standardized format for exchanged messages.

SOAP response

```
<?xml version="1.0" encoding="utf-8"?>
<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

  <soapenv:Body>
    <doItResponse xmlns="http://DefaultNamespace"/>
  </soapenv:Body>

</soapenv:Envelope>
```

19

Here is the SOAP response.

REST Web Services (2000++)

- A simplified version, not a standard, rather a style (of use of simple features of the web)
- Increasingly popular
- Use of HTTP methods
 - GET : get data from the server
 - POST : create data in the server
 - PUT : update data in the server
 - DELETE : delete data in the server
- Invoked service encoded in the URL
 - http://<machine>/module/service
 - For instance
 - HTTP request to <machine>
 - GET /module/service
 - service only returns data from the server

20

SOAP/WSDL based WS were very popular some years ago. They are becoming obsolete.

An evolution of WS is REST WS. This is a simplified version which is very popular now. Notice that REST WS is not a standard, but rather a recommendation or a style of implementation, based on simple features of the web.

It relies on HTTP requests (GET, POST, PUT, DELETE), but mainly GET and POST are used. GET is used when you want to read (only) data from the WS while POST is used when you want to modify something in the WS.

The service that you call is encoded in the URL.

For instance a GET HTTP request on URL http://<machine>/module/service

This service is supposed to return data from the server, without updating anything in the server (its a style, it is not enforced).

REST Web Services

- Parameter passing
 - HTTP parameters
 - Format : field1=value1&field2=value2
 - GET : parameters in the URL
 - `http://nom_du_serveur/cgi-bin/script.cgi?champ1=valeur1&...`
 - POST : parameters are included in the body of the HTTP request
 - XML or JSON (or any other)
 - Included in the body of the HTTP request
- Response
 - Included in the body of the HTTP response: XML or JSON (or any other)
- Description of a REST WS (available on the net)
 - A simple document describing the methods and parameters
 - No WSDL

21

Parameter passing can be based on HTTP parameters. With HTTP parameters, parameters are encoded as a String `field1=value1&field2=value2 ...`

This parameter string is passed in the URL if you use the GET HTTP method, and it is passed in the body of the request if you use the POST HTTP method.

You can also pass parameters in a document (XML or JSON or any format) in the body of the request.

A service can return a document (XML or JSON) in the body of the response (which corresponds to return parameters).

The description of a REST WS is simply a document describing the services that you may call and the passed parameters (names, formats).

Example of existing REST WS

Currency API Request
The base URL for our currency API is
`https://www.amdoren.com/api/currency.php`

Request Parameters

Parameter	Description
api_key	Your assigned API key. This parameter is required.
from	The currency you would like to convert from. This parameter is required.
to	The currency you would like to convert to. This parameter is required.
amount	The amount to convert from. This parameter is optional. Default is a value of 1.

Example:
To get the latest exchange rate in EUR for 1 USD:
`https://www.amdoren.com/api/currency.php?api_key=1B2zdLmH2yCYaXjgT26x6fom=USDto=EUR`

Currency API Response

Element	Description
error	Error code. Value greater than zero indicates an error. See list below.
error_message	Short description of the error. See list below.
amount	The exchange rate or amount converted.

Example:
JSON data returned from our currency API request:
{ "error" : 0, "error_message" : "-", "amount" : 0.90168 }

22

Here is an example of description of a REST WS. This is for a currency converter.

It says that you have one service available :

<https://www.amdoren.com/api/currency.php>

It lists the parameters that may be passed in the HTTP GET request. A example is given.

It then describes the response which is a JSON. A example is given.

Development of REST WS

- You can develop your application by hand in any programming language
 - Verbose and error prone
 - As for RPC, code can be generated
- Many development environments
 - e.g. Resteasy and Jersey
 - Resteasy in the following of the talk (client and server sides)
 - A view on Spring (mainly server side)

23

Existing REST WS (client with resteasy)

Interface

```
@Path("/")
public interface ServiceInterface {

    @GET
    @Path("/currency.php")
    @Produces({ "application/json" })
    public Result convert(@QueryParam("api_key") String key, @QueryParam("from") String from,
        @QueryParam("to") String to);

}
```

Java bean (generated from JSON)

```
public class Result {

    String error;
    String error_message;
    String amount;

    // getters/setters

}
```

24

Many development environments can be used to REST WS development.

In the following, we overview the used of resteasy (on the server side and the client side) and we have a look at Spring. Both will be used in the labs.

With the currency converter, as said in the documentation, the conversion method takes 3 HTTP parameters (api_key, from, to, the last is optional) and it returns a JSON.

The 3 HTTP parameters are associated with Java parameters (with @QueryParam) and a Java bean is created for the JSON.

Existing REST WS (client with resteasy)

Client

```
public class Client {  
    public static void main(String args[]) {  
        final String path = "https://www.amdoren.com/api";  
  
        ResteasyClient client = new ResteasyClientBuilder().build();  
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));  
        ServiceInterface proxy = target.proxy(ServiceInterface.class);  
  
        Result r = proxy.convert("9xwjRjxTtnzuGKH7LcWC5Vengr52F3", "EUR", "AMD");  
        System.out.println("convert: "+r.getError()+"/"+r.getError_message()+  
            +"/"+r.getAmount());  
    }  
}
```

easy
invocation

25

And here is an example of client which invokes the service.

Implementing a service with resteasy

WS class

```
@Path("/")  
public class Facade {  
    static Hashtable<String, Person> ht = new Hashtable<String, Person>();  
  
    @POST  
    @Path("/addperson")  
    @Consumes({ "application/json" })  
    public String addPerson(Person p) {  
        ht.put(p.getId(), p);  
        return "person added";  
    }  
  
    @GET  
    @Path("/getperson")  
    @Produces({ "application/json" })  
    public Person getPerson(@QueryParam("id") String id) {  
        return ht.get(id);  
    }  
  
    @GET  
    @Path("/listpersons")  
    @Produces({ "application/json" })  
    public Collection<Person> listPersons() {  
        return ht.values();  
    }  
}
```

Person is a simple POJO

26

As for SOAP/WS, many tools were implemented to help developers.

Here, we present Resteasy (Jersey is also a very popular one you may look at).

On the server side, you can use annotations in a Java program to say :

- each method is associated with a path in the URL used to access the WS
- @Path : specifies the element of the path associated with the class or the method. Here method addPerson() is associated with path /addperson
- @POST or @GET : specifies which HTTP method is used. Notice that GET returns an object (data) while POST returns an HTTP code (and a message).
- @Consumes : specifies that we receive a JSON object which is deserialized into a Java object.
- @Produces : specifies that we return a Java object which is serialized into a JSON object.
- @QueryParam : the getPerson() method has an "id" parameter. The QueryParam annotation associates this parameter with an "id" HTTP parameter.

Implementing a service with resteasy

- Add the RestEasy jars in Tomcat (lib)
- In eclipse (not easy with vscode)
 - Create a Dynamic Web Project
 - Add RestEasy jars in the buildpath
 - Create a package
 - Implement the WS classes (Facade + Person)
 - Add a class RestApp

```
public class RestApp extends Application {  
    private Set<Object> singletons = new HashSet<Object>();  
    public RestApp() {  
        singletons.add(new Facade());  
    }  
    public Set<Object> getSingletons() {  
        return singletons;  
    }  
}
```

27

To run this example :

- add the Resteasy jars in Tomcat and Eclipse
- create a dynamic web project (a servlet project)
- add the RestEasy jars in the buildpath of the project
- create a package and the previously developed classes
- add the RestApp class

Implementing a service with resteasy

- Add a web.xml descriptor in the WebContent/WEB-INF folder

```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  
        http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">  
    <servlet>  
        <servlet-name>resteasy-servlet</servlet-name>  
        <servlet-class>  
            org.jboss.resteasy.plugins.server.servlet.HttpServletDispatcher  
        </servlet-class>  
        <init-param>  
            <param-name>javax.ws.rs.Application</param-name>  
            <param-value>pack.RestApp</param-value>  
        </init-param>  
    </servlet>  
    <servlet-mapping>  
        <servlet-name>resteasy-servlet</servlet-name>  
        <url-pattern>/*</url-pattern>  
    </servlet-mapping>  
</web-app>
```

- Export the war in Tomcat

28

Add the web.xml descriptor in the WebContent/WEB-INF folder and export a war (in the webapps folder of Tomcat)

Publish the WS

- Just write a documentation which says that
 - The WS is available at <http://<machine-name>:8080/<project-name>/>
 - Method addperson with POST receives a person JSON :

```
{
  "id": "00000",
  "firstname": "Alain",
  "lastname": "Tchana",
  "phone": "0102030405",
  "email": "alain.tchana@enseeiht.fr"
}
```

- Method getperson with GET receives an HTTP parameter id and returns a person JSON
 - Method listperson returns a JSON including a set of persons
- A caller may use any tool (not only RestEasy)

29

Implementing the client with resteasy

- From the previous documentation, a client can write the interface

```
@Path("/")
public interface FacadeInterface {

    @POST
    @Path("/addperson")
    @Consumes({ "application/json" })
    public String addPerson(Person p);

    @GET
    @Path("/getperson")
    @Produces({ "application/json" })
    public Person getPerson(@QueryParam("id") String id);

    @GET
    @Path("/listpersons")
    @Produces({ "application/json" })
    public Collection<Person> listPersons();
}
```

30

Publication of a REST WS is simply a document describing the interface.

On the client side, from the documentation, a user can write a Java interface with Resteasy annotations. Of course, it's very similar to what we wrote on the server side, but we could do it for a WS we don't know (we only have the documentation).

Implementing the client with resteasy

- And write a class which invokes the WS

```
public class TestClient {  
    public static void main(String[] args) {  
        final String path = "http://localhost:8080/rest-server";  
        ResteasyClient client = new ResteasyClientBuilder().build();  
        ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));  
        FacadeInterface proxy = target.proxy(FacadeInterface.class);  
  
        String resp;  
        resp = proxy.addPerson(new Person("007", "James Bond"));  
        System.out.println(resp);  
        resp = proxy.addPerson(new Person("006", "Dan Hagi"));  
        System.out.println(resp);  
  
        System.out.print("List Person: ");  
        Collection<Person> l = proxy.listPersons();  
        for (Person p : l) System.out.print "[" + p.getId() + "/" + p.getName() + " ]";  
        System.out.println();  
  
        Person p = proxy.getPerson("006");  
        System.out.println("get Person: " + p.getId() + "/" + p.getName());  
    }  
}
```

31

The previous annotated interface (FacadeInterface) makes it easy to invoke the service. We can build a proxy object of type FacadeInterface.

This proxy allows programming service invocations simply as method calls.

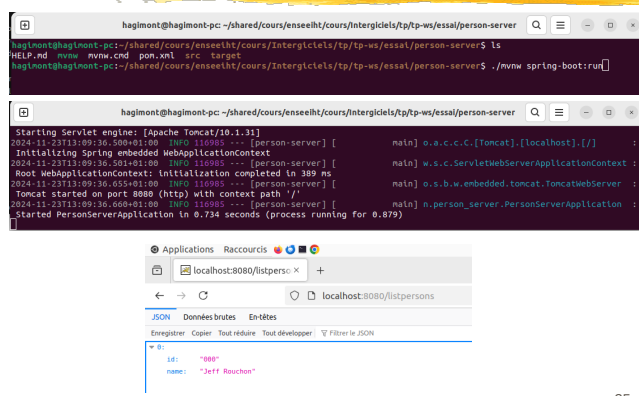
Implementing the client with resteasy

- In eclipse or vscode
 - Create a Java Project
 - Add RestEasy jars in the project
 - Implement the Java bean that correspond to the JSON
 - Automatic generation with <https://www.site24x7.com/tools/json-to-java.html>
 - Implement the interface and the client class (FacadeInterface + TestClient)
 - Run

32

This is the procedure to run the client.

Run as a standalone application (maven is used to run the app, including Tomcat)



The image shows a terminal window and a web browser. The terminal window displays the output of the `./mvnw spring-boot:run` command, showing the application starting on port 8080. The web browser shows the `localhost:8080/listpersons` endpoint, displaying a JSON response with a single person object.

```
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server$ ls
HELP.md mvnw mvnw.cmd pom.xml src target
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server$ ./mvnw spring-boot:run

Starting Servlet engine: [Apache Tomcat/10.1.31]
2024-11-23T13:09:36.006+0100 [INFO] 116905 --- [person-server] [main] o.a.c.c.C.[Tomcat].[localhost].[/]
Initializing Spring embedded WebApplicationContext
2024-11-23T13:09:36.201+0100 [INFO] 116905 --- [person-server] [main] w.s.c.ServletWebServerApplicationContext :
Root WebApplicationContext: initialization completed in 389 ms
2024-11-23T13:09:36.655+0100 [INFO] 116905 --- [person-server] [main] o.s.b.w.embedded.tomcat.TomcatWebServer :
Tomcat started on port 8080 (http) with context path '/'
2024-11-23T13:09:36.666+0100 [INFO] 116905 --- [person-server] [main] n.person_server.PersonServerApplication :
Started PersonServerApplication in 0.734 seconds (process running for 0.879)

Applications Raccourcis
localhost:8080/listperso x +
localhost:8080/listpersons
JSON Données brutes Entêtes
Enregistrer Copier Tout réduire Tout développer Filtrer la JSON
0:
{
  id: "000",
  name: "Jeff Rauchon"
}
```

35

In you VScode project, you have a "mvnw" script. It's a Maven script.

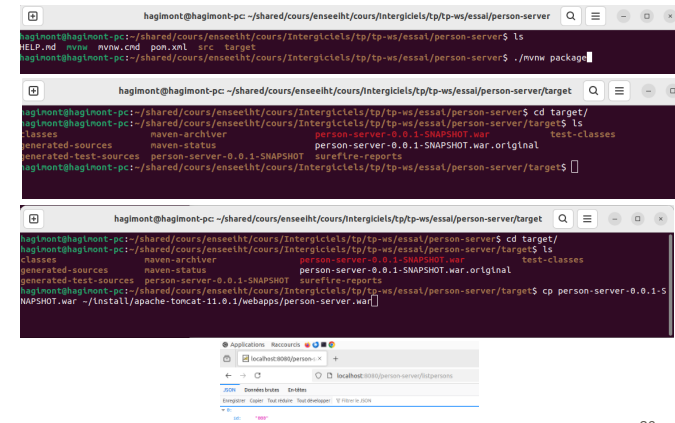
If you run: `./mvnw spring-boot:run`

It will start a Tomcat server which executes your REST WS. Thanks to Maven, it downloads all the dependencies.

You can test your WS with a web browser.

Notice that the URL (`http://localhost:8080/listpersons`) used in the browser only includes the invoked method name.

Export a war in Tomcat



The image shows a terminal window and a web browser. The terminal window displays the output of the `./mvnw package` command, showing the application packaging a war file. The web browser shows the `localhost:8080/person-server/listpersons` endpoint, displaying a JSON response with a single person object.

```
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server$ ls
HELP.md mvnw mvnw.cmd pom.xml src target
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server$ ./mvnw package

hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server$ cd target/
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server/target$ ls
classes maven-archiver maven-status person-server-0.0.1-SNAPSHOT.war original test-classes
generated-sources person-server-0.0.1-SNAPSHOT surefire-reports
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server/target$

hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server/target$ cd target/
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server/target$ ls
classes maven-archiver maven-status person-server-0.0.1-SNAPSHOT.war original test-classes
generated-sources person-server-0.0.1-SNAPSHOT surefire-reports
hagimont@hagimont-pc: ~/shared/cours/enseelht/cours/Intergiciels/tp/tp-ws/essal/person-server/target$ cp person-server-0.0.1-S
NAPSHOT.war ~/install/apache-tomcat-11.0.1/webapps/person-server.war

Applications Raccourcis
localhost:8080/person-s x +
localhost:8080/person-server/listpersons
JSON Données brutes Entêtes
Enregistrer Copier Tout réduire Tout développer Filtrer la JSON
0:
{
  id: "000",
  name: "Jeff Rauchon"
}
```

36

Another way to run your REST WS is to export a war archive that you deploy in a Tomcat server (already running).

In your VScode project, you can run: `./mvnw package`

It will produce a war in the target folder (`person-server-0.0.1-SNAPSHOT.war`).

Here, I copy this war into my Tomcat (webapps folder).

I rename it as `person-server.war` as the name of the war is used in the URL used to access the WS.

The URL (`http://localhost:8080/person-server/listpersons`) used in the browser includes the name of the war.

Invoking a rest API from Spring

```
1: @RestController
2: public class ClientController {
3:
4:     final static String path = "http://localhost:8080/person-server/";
5:
6:     @GetMapping("/call")
7:     public Collection<Person> call() {
8:
9:         try {
10:
11:             RestClient client = RestClient.create();
12:             client.post().uri(path + "addperson").contentType(MediaType.APPLICATION_JSON).body(new Person(id="1111", name="Dan Hugi")).retrieve().toResponseBody();
13:             Collection<Person> list = client.get().uri(path + "listpersons").retrieve().body(new ParameterizedTypeReference<>() {});
14:             return list;
15:         } catch (Exception ex) {
16:             ex.printStackTrace();
17:             return null;
18:         }
19:     }
20: }
```

37

Spring also provides support for invoking external REST WS from within a Spring WS.

The class to use for that is `RestClient`.

Here, I programmed a simple Spring REST WS with a method "call". In that method, I invoke methods "addperson" and "listpersons" from the previous service.

Conclusion

- Web Services: a RPC over HTTP, exchanging XML or JSON
- Interesting for heterogeneity as there are tools in all environments
- Recently
 - SOAP WS less used
 - REST + XML/JSON more popular
 - Micro-services: used for structuring backend applications

38

To conclude, Web services aim at implementing a RPC service on top of HTTP and relying on standard formats (XML, JSON).

One of the main interest is the independence between the server (the service provider) and the client (the service consumer). They can be from different organizations and use different tools, OS, or languages.

The recent evolution is an obsolescence of SOAP and an increased popularity of REST and JSON.

Recently, the micro-service architecture was proposed. It consists in architecting large applications (especially backend applications) in terms of a set of interconnected REST WS (components). The advantage is independence between components.