Web Services

Daniel Hagimont

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

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

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 retuns

```
<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

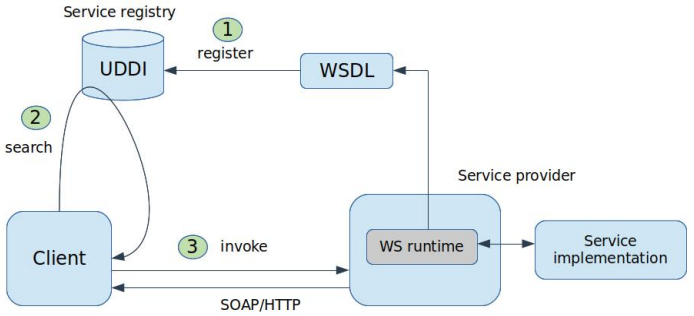
XML-RPC was a precursor of what are web services now.

XML-RPC was a RPC protocol relying of XML for the representation of requests and HTTP for the transport of requests.

The idea was to be independent from execution platforms or languages and to rely on widely recognized and adopted formats.

XML-RPC was a precursor and evolved into SOAP, the protocol used in web services.

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

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 used 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.

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 langage environments

6

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 received 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

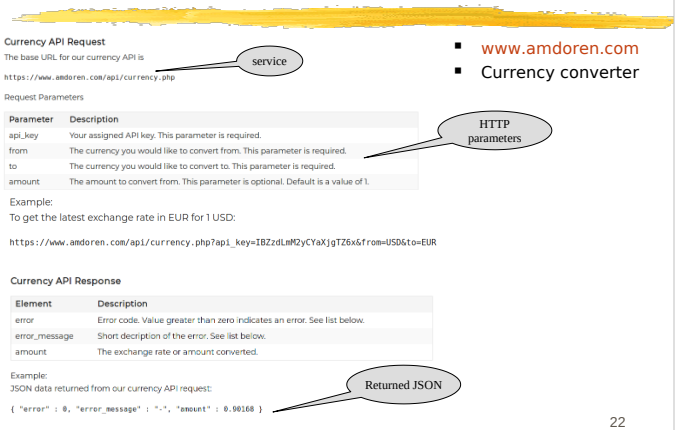
- Eclipse JEE
- Apache Axis
- Creation of a Web Service
  - From a Java class
  - In the Tomcat runtime
  - Generation of the WSDL file
- Creation of a client application
  - Generation of stubs from a WSDL file
  - Programming of the client

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.

Example of existing REST WS



- www.amdoren.com
- Currency converter

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

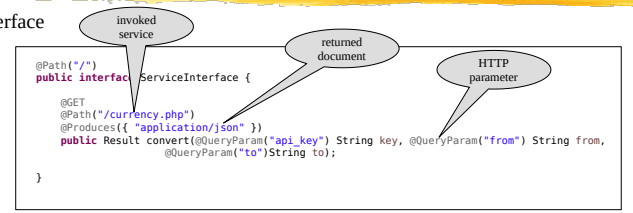
- You can develop your application by hand in any programming langage
  - Verbose and error prone
  - As for RPC, code can be generated
- Many development environments
  - e.g. Resteasy and Jersey
  - Resteasy in the following of the talk (client and server sides)
  - A view on Spring (mainly server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

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.

Existing REST WS (client with resteasy)

Interface



Java bean (generated from JSON)

```
public class Result {

    String error;
    String error_message;
    String amount;

    // getters/setters
```


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

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();
    }
}
```

Receives a JSON
Deserialized into a Java object
Returns a String

Returns an object
Serialized into a JSON
Receives an id HTTP parameter

Person is a simple POJO

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;
          }
}
```


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">
  <display-name>essai-server</display-name>
  <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

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)

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

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();
}
```


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());
    }
}
```


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

This is the procedure to run the client.

A view on Spring

- A development environment for server side
  - Spring-boot: facilitate the configuration
    - Relies on Maven (dependencies)
    - Can produce
      - A standalone application (including Tomcat)
      - A war to be deployed in Tomcat
    - Also provides client side support (within a Spring server)
  - In VScode
    - Extension: Spring initializr java support

Spring is a development environment for developing REST WS.

Spring-boot is an extension which simplifies the configuration. It relies on Maven.

Thanks to Spring-boot, you can produce:

- an application which embeds Tomcat. When you launch it, you start a Tomcat web server which included your REST WS application.

- a war archive which includes your REST WS application. This war can be deployed in a running Tomcat server.

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

In VScode, you can use the "Spring initializr" extension which automates the creation of a Spring-boot project.

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

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 architecturing large applications (especially backend applications) in terms of a set of interconnected REST WS (components). The advantage is indenpendence between components.