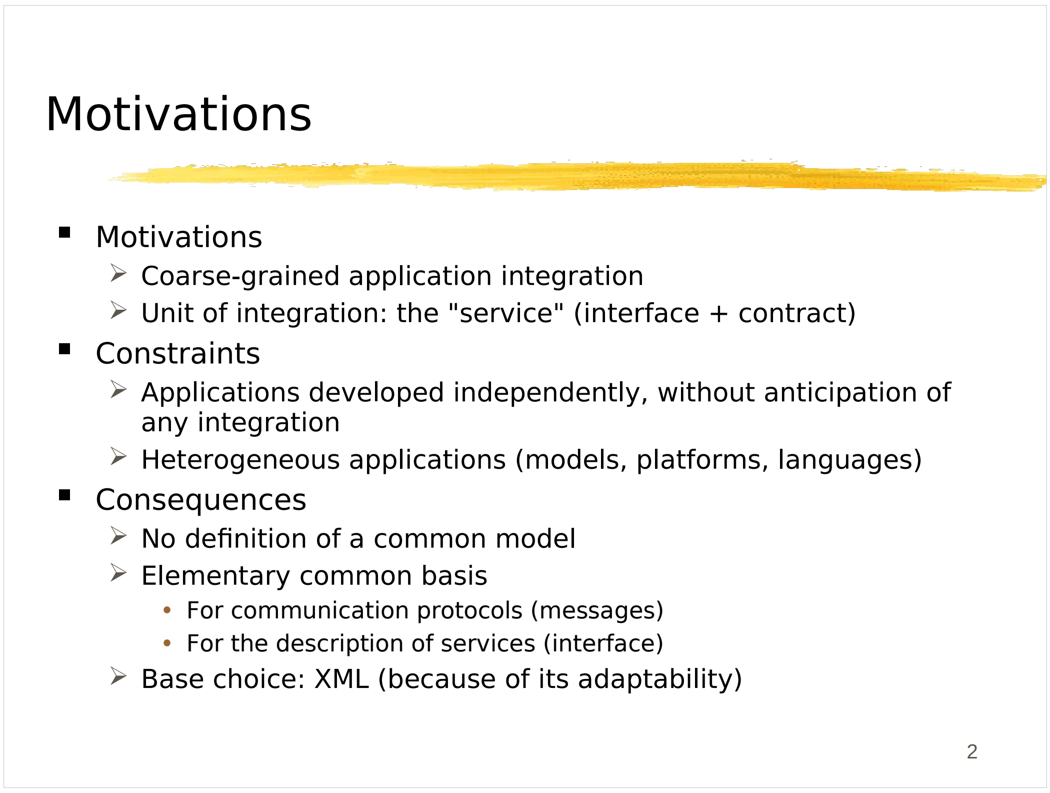
Web Services

Daniel Hagimont

IRIT/ENSEEIHT
2 rue Charles Camichel - BP 7122
31071 TOULOUSE CEDEX 7

Daniel.Hagimont@enseeiht.fr
http://hagimont.perso.enseeiht.fr

This lecture is about web services.

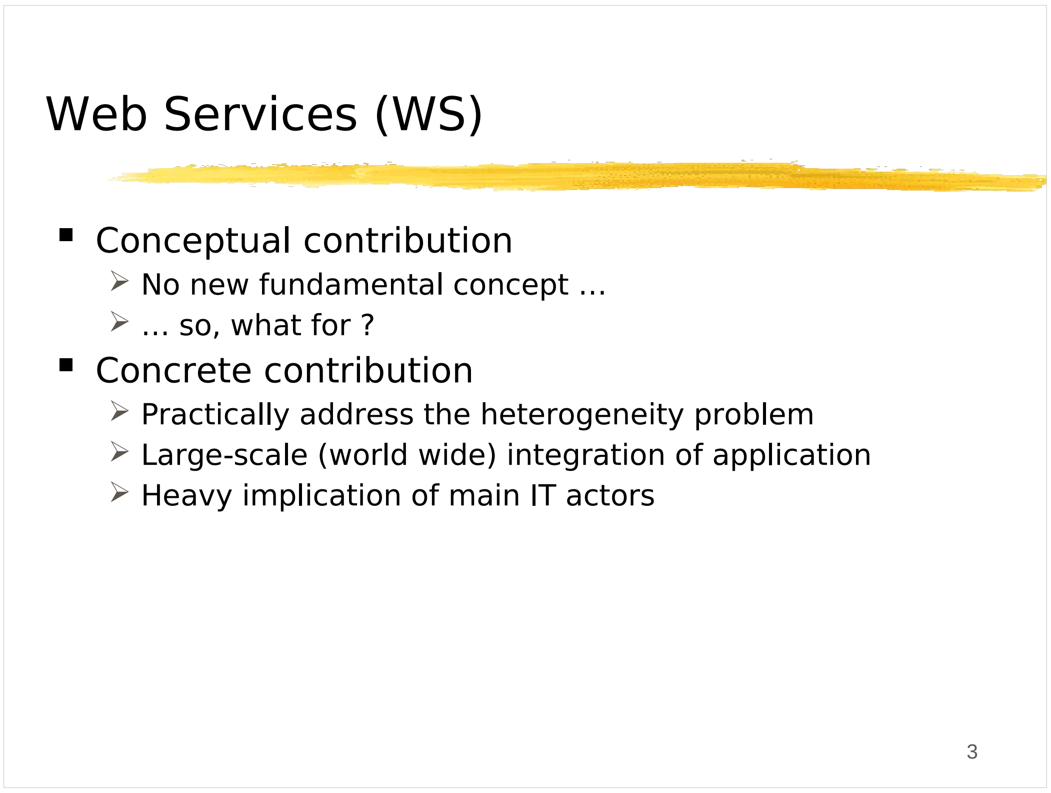


Motivations

- 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
  - No definition of a common model
  - Elementary common basis
    - For communication protocols (messages)
    - For the description of services (interface)
  - Base choice: XML (because of its adaptability)

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.



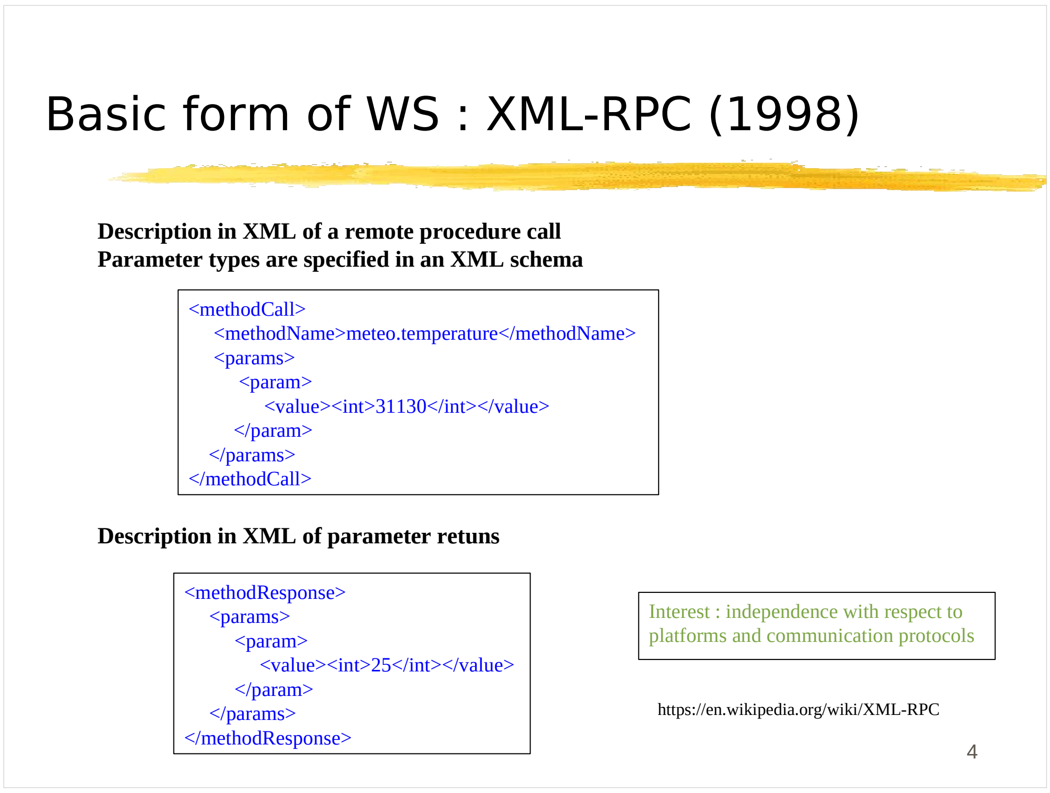
Web Services (WS)

- Conceptual contribution
  - No new fundamental concept …
  - … so, what for ?
- Concrete contribution
  - Practically address the heterogeneity problem
  - Large-scale (world wide) integration of application
  - Heavy implication of main IT actors

No new conceptual concepts here, but justified by

- it addresses the problem of heterogeneity. Providers and consumers may be of different organizations and use different languages, OS …

- it was pushed by the main IT actors



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

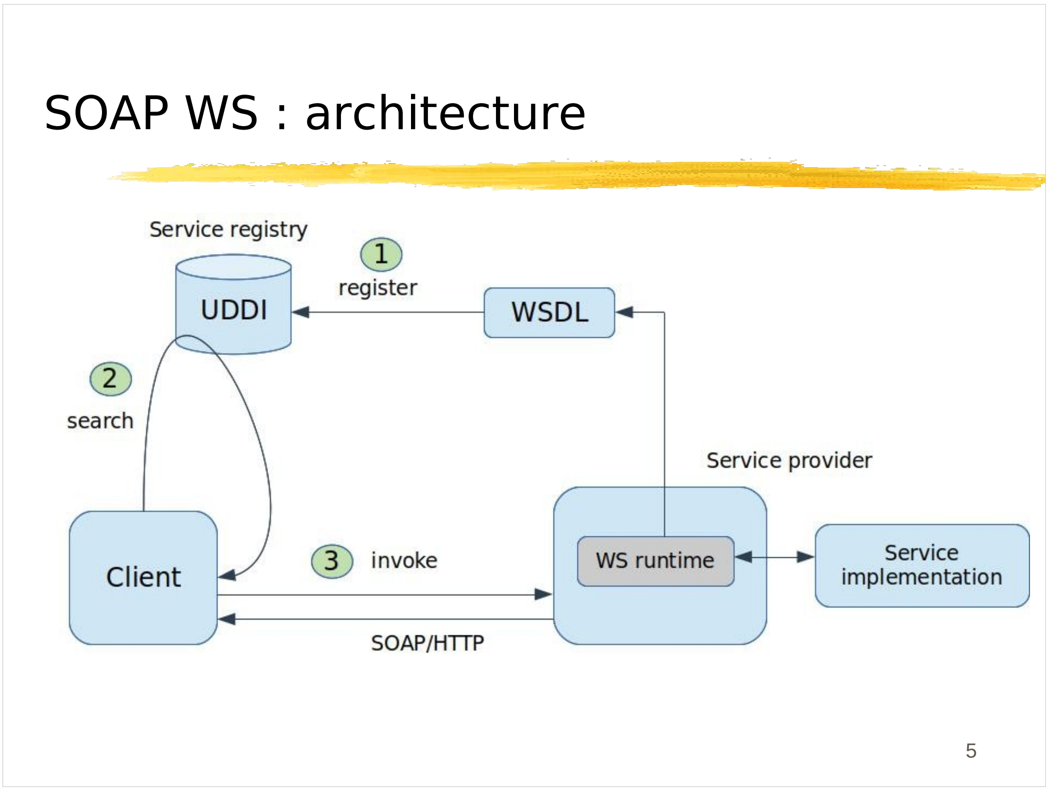
https://en.wikipedia.org/wiki/XML-RPC

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

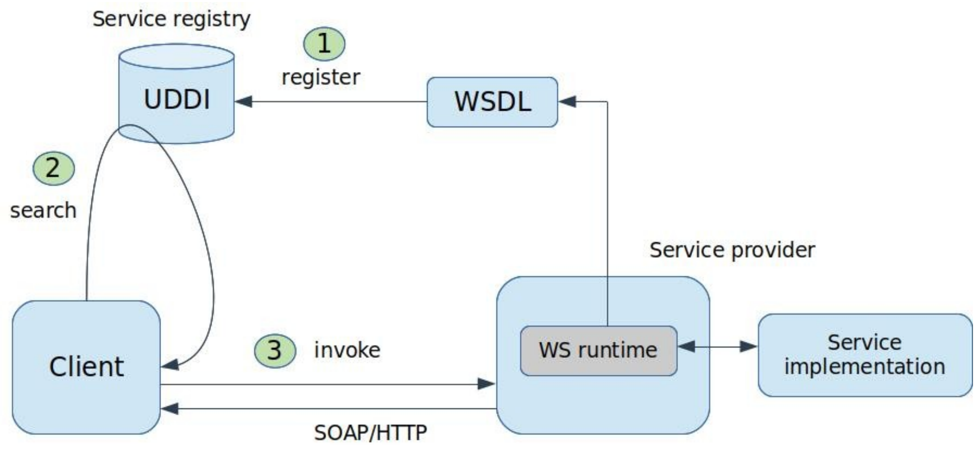
XML-RPC was a RPC protocol relying of XML for the representation of requests and HTTP for the transport of requests.

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



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 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.

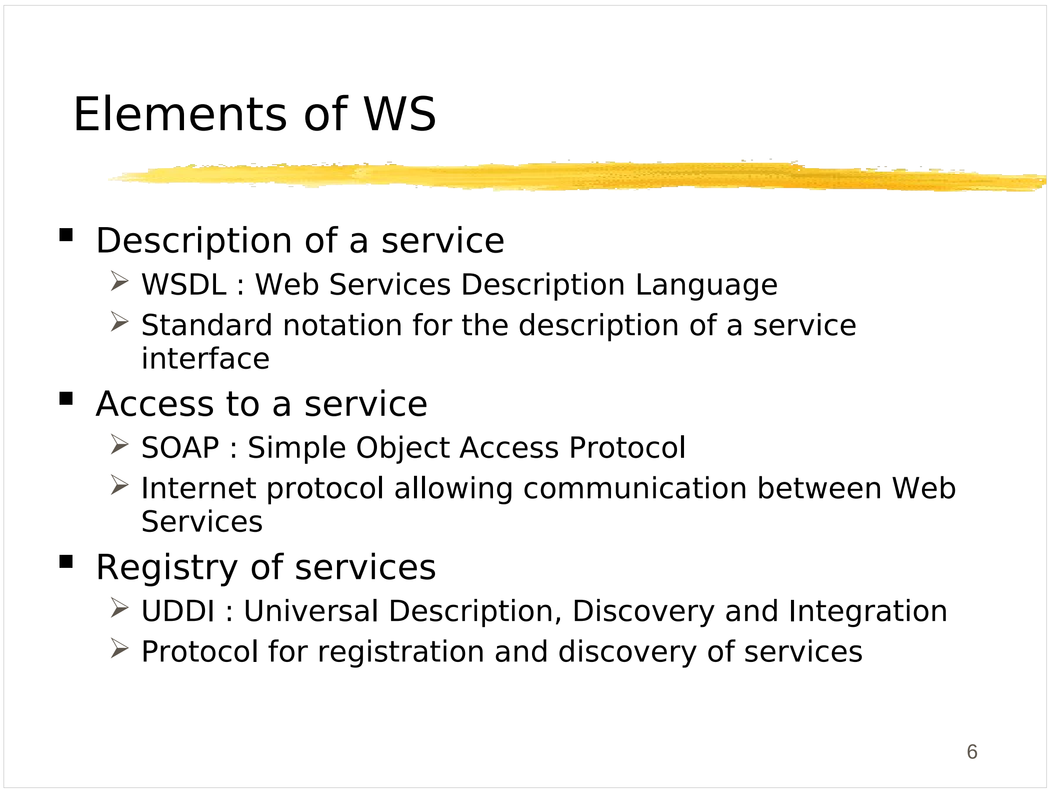
The WS architecture specifies that a service registry (a naming service) should be used for the publication and discovery of WSDL descriptions. However, UDDI was not actually used. Generally the WSDL of a WS can be published on a Web server as any document.

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 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
  - Internet protocol allowing communication between Web Services
- Registry of services
  - UDDI : Universal Description, Discovery and Integration
  - Protocol for registration and discovery of services

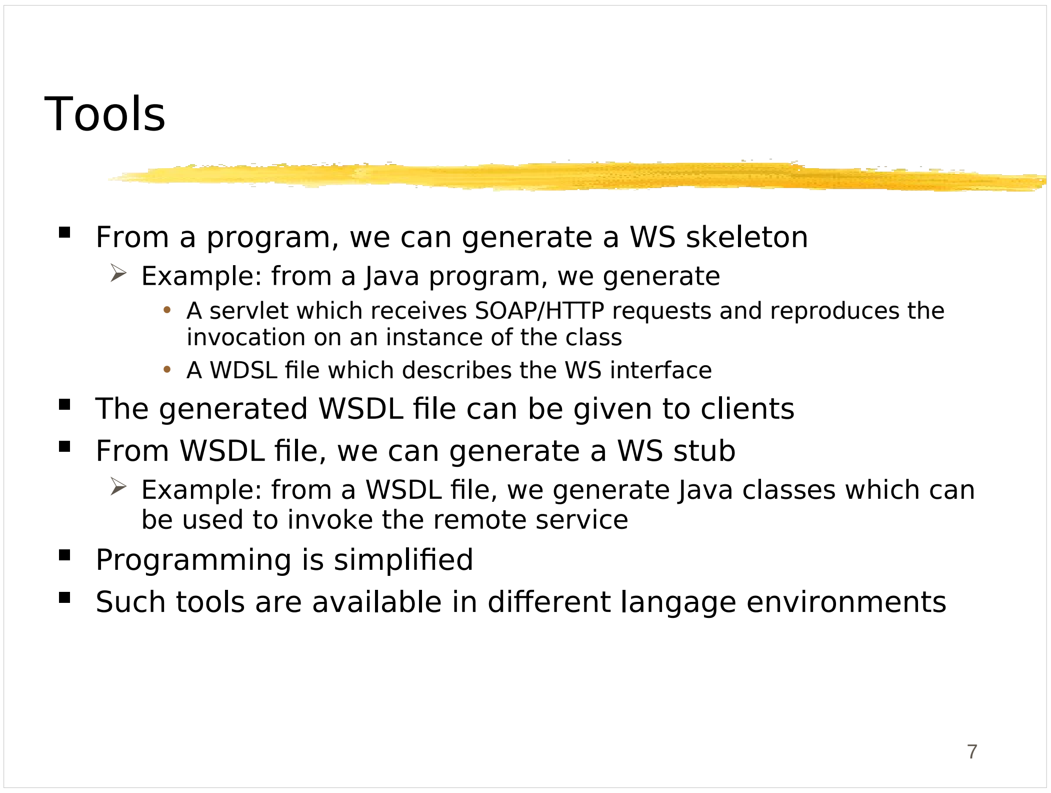
6

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 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

7

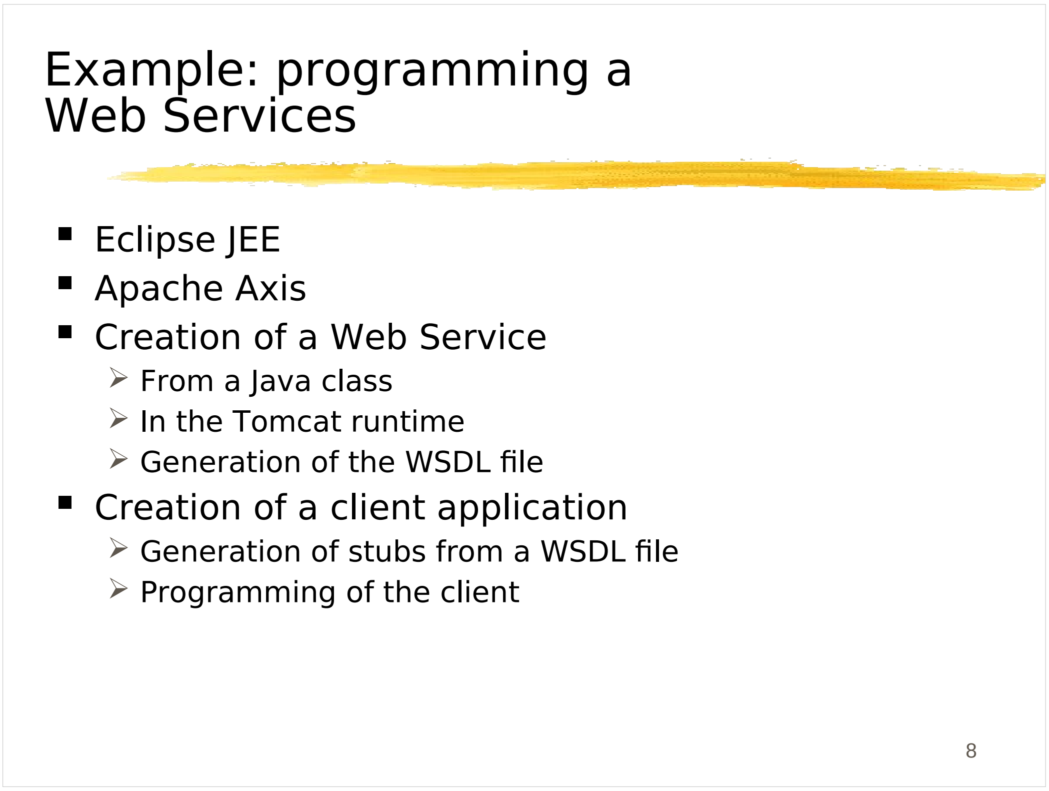
To illustrate this, we give an example of use in the Java environment.

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 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

8

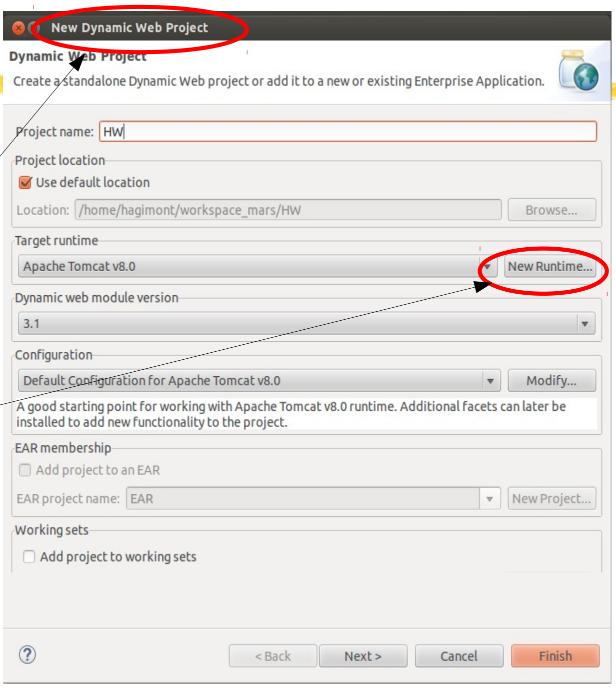
We use Eclipse JEE and Apache Axis which is available in Eclipse JEE.

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



New Dynamic Web Project

Dynamic Web Project

Create a standalone Dynamic Web project or add it to a new or existing Enterprise Application.

Project name: HW

Project location

[✓] Use default location

Location: /home/hagimont/workspace_mars/HW

Target runtime: Apache Tomcat v8.0

Dynamic web module version: 3.1

Configuration: Default Configuration for Apache Tomcat v8.0

A good starting point for working with Apache Tomcat v8.0 runtime. Additional facets can later be installed to add new functionality to the project.

EAR membership

[ ] Add project to an EAR

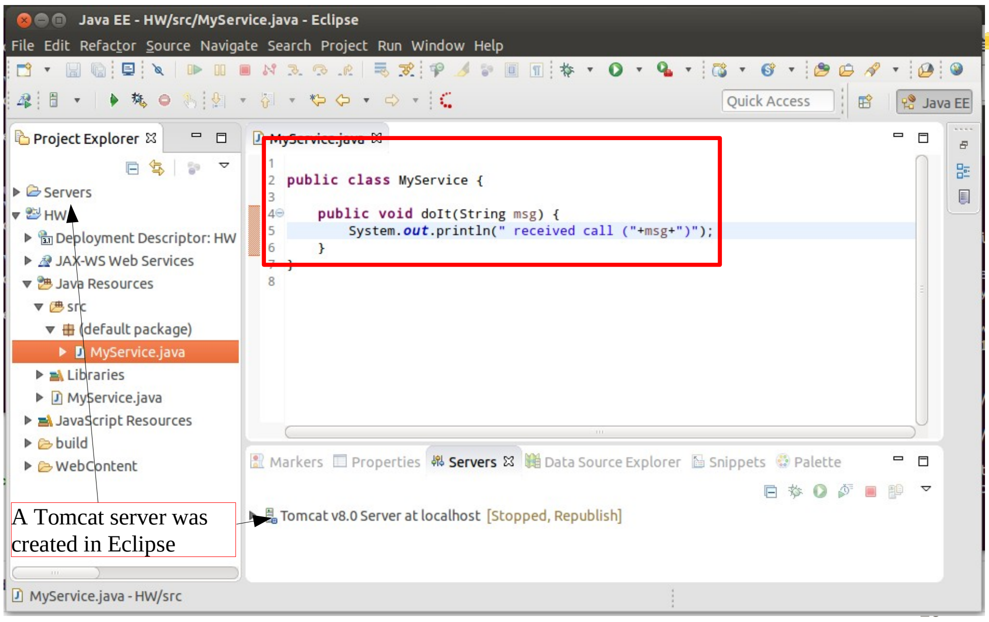
EAR project name: EAR

Working sets

[ ] Add project to working sets

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

Create a Class



```
public class MyService {

    public void doIt(String msg) {
        System.out.println(" received call ("+msg+")");
    }
}
```

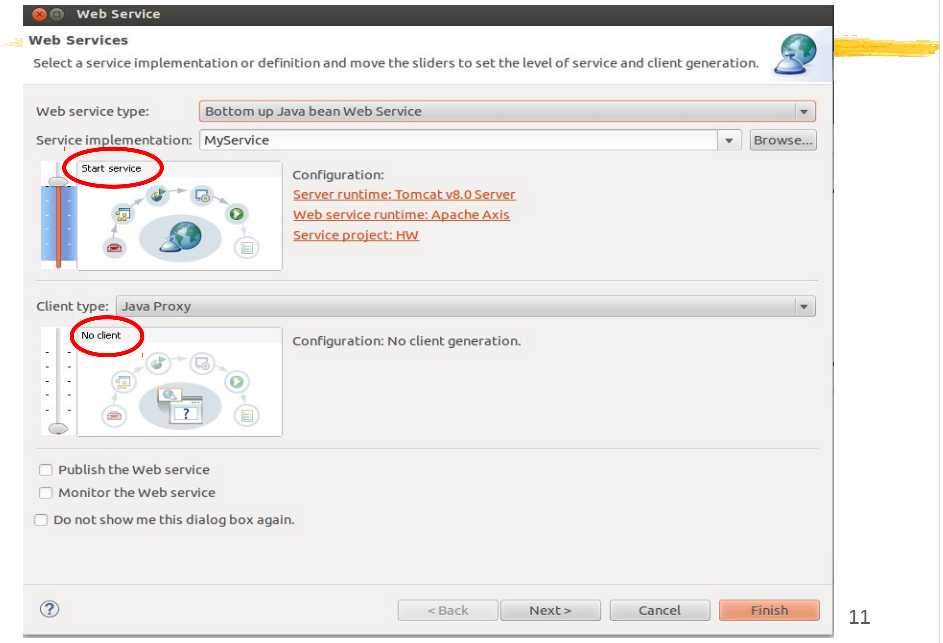
Tomcat v8.0 Server at localhost [Stopped, Republish]

A Tomcat server was created in Eclipse

In this project, we create a class.

Notice that a Tomcat server is running in Eclipse.

From source file : Web Service → create Web Service



Web Service

Web Services

Select a service implementation or definition and move the sliders to set the level of service and client generation.

Web service type: Bottom up Java bean Web Service

Service implementation: MyService

Start service

Configuration:
Server runtime: Tomcat v8.0 Server
Web service runtime: Apache Axis
Service project: HW

Client type: Java Proxy

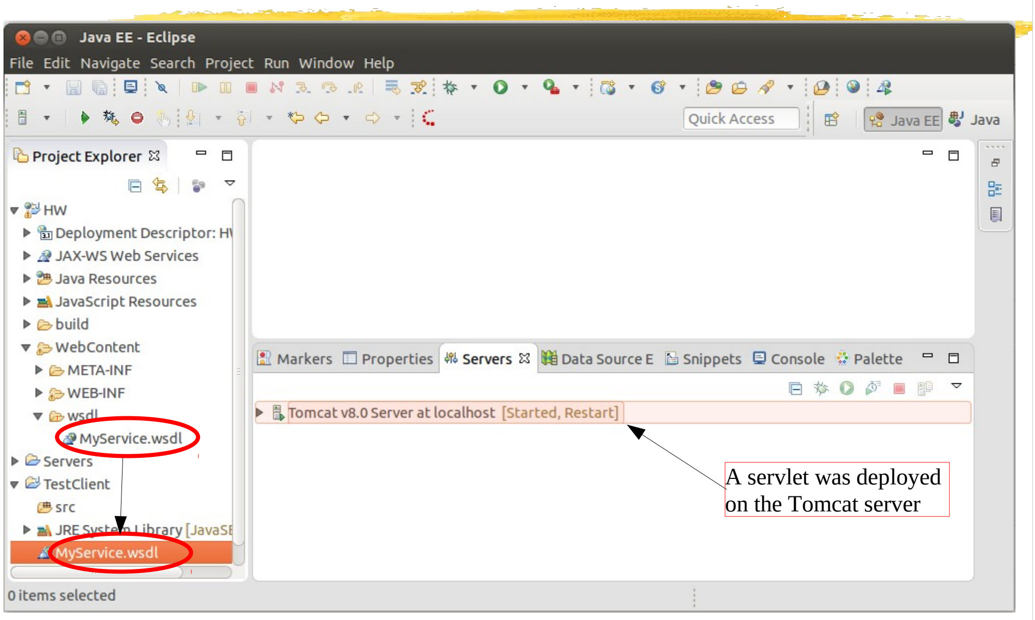
No client

Configuration: No client generation.

- [ ] Publish the Web service
- [ ] Monitor the Web service
- [ ] Do not show me this dialog box again.

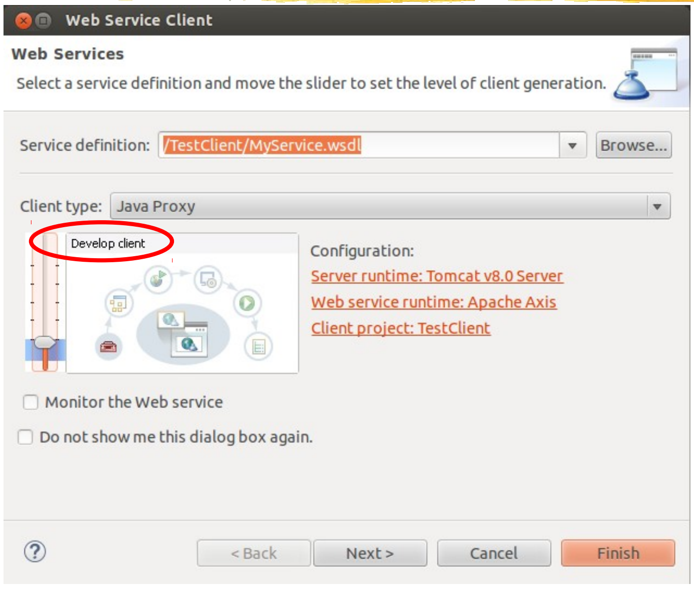
From the source file of the class, we can generate (right click) a WS from this file.

Copy the generated WSDL file in a new Java project



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

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


Web Service Client

Web Services

Select a service definition and move the slider to set the level of client generation.

Service definition: /TestClient/MyService.wsdl

Client type: Java Proxy

Configuration:

Server runtime: Tomcat v8.0 Server

Web service runtime: Apache Axis

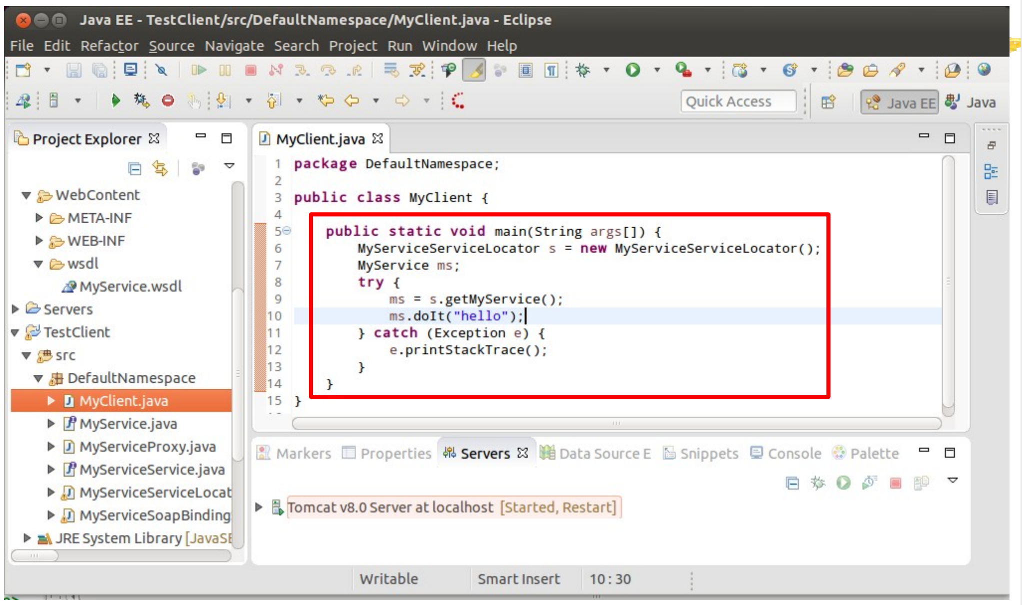
Client project: TestClient

☐ Monitor the Web service

☐ Do not show me this dialog box again.

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

Program a client



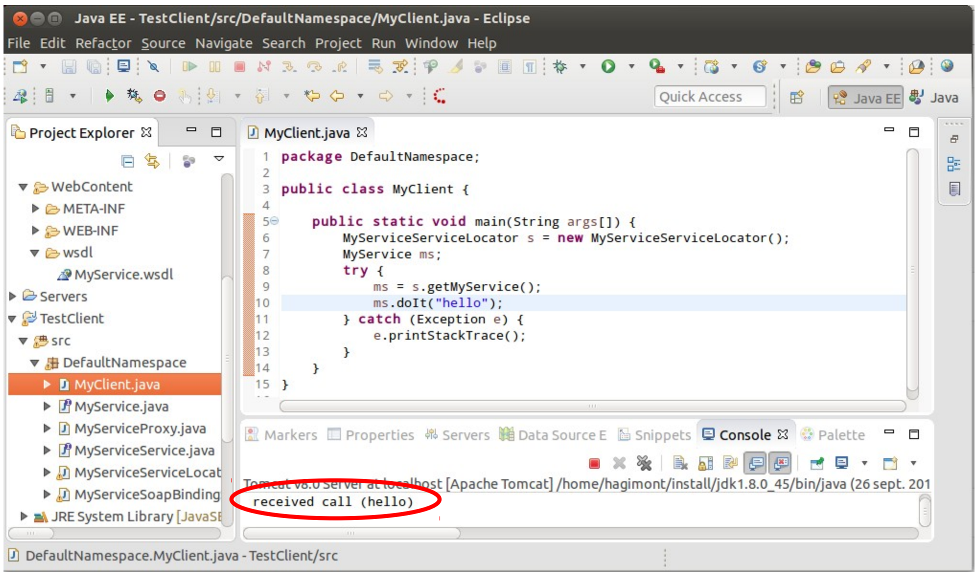
```
package DefaultNamespace;

public class MyClient {

    public static void main(String args[]) {
        MyServiceServiceLocator s = new MyServiceServiceLocator();
        MyService ms;
        try {
            ms = s.getMyService();
            ms.doIt("hello");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Tomcat v8.0 Server at localhost [Started, Restart]

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).

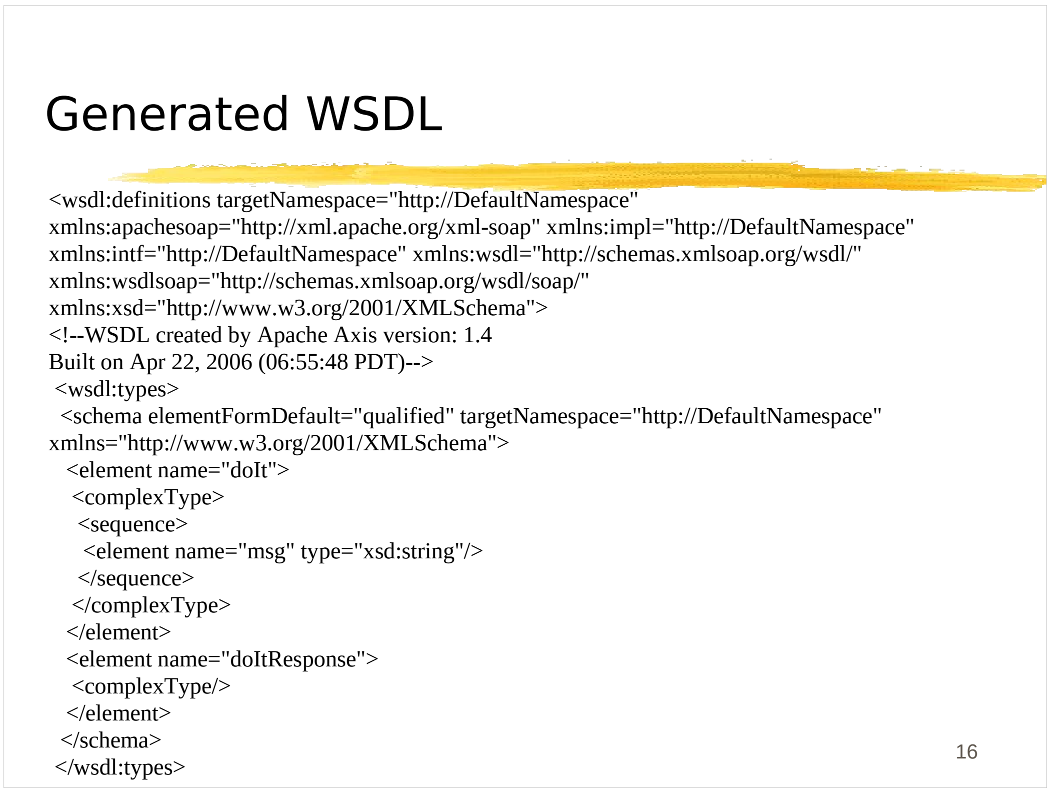
Run



```java
package DefaultNamespace;

public class MyClient {

    public static void main(String args[]) {
        MyServiceServiceLocator s = new MyServiceServiceLocator();
        MyService ms;
        try {
            ms = s.getMyService();
            ms.doIt("hello");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```
Tomcat v8.0 Server at localhost [Apache Tomcat] /home/hagimont/install/jdk1.8.0_45/bin/java (26 sept. 201
 received call (hello)
```

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

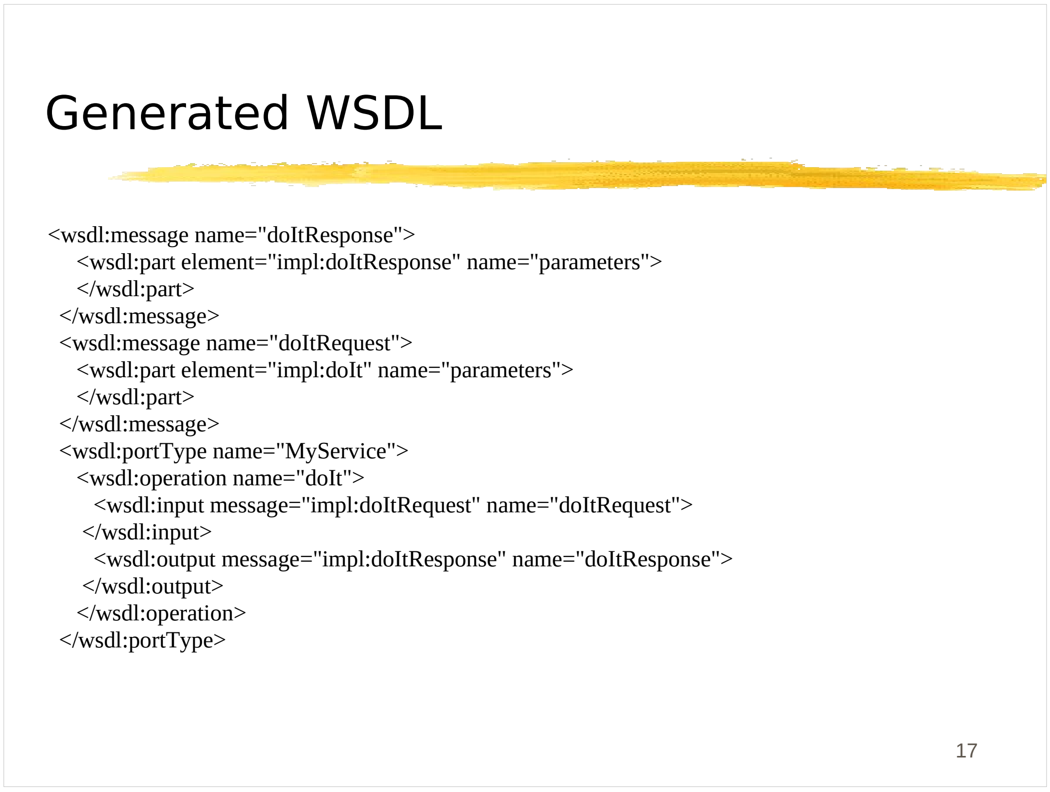


Generated WSDL

```
<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>
```

We can have a look at the WSDL description.

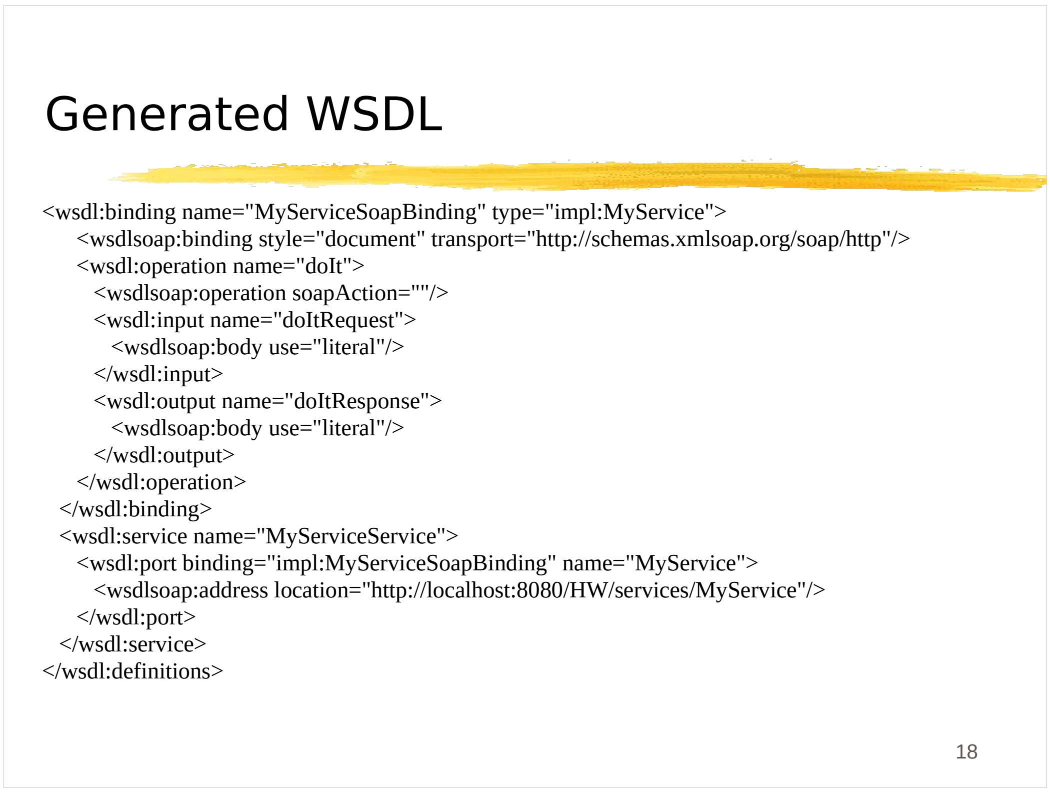
We can see that the WSDL syntax is not very simple. Therefore such WSDL descriptions are not written by the user, but generally generated by the tool on the server side and imported by the client.



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>
```

Very verbose !

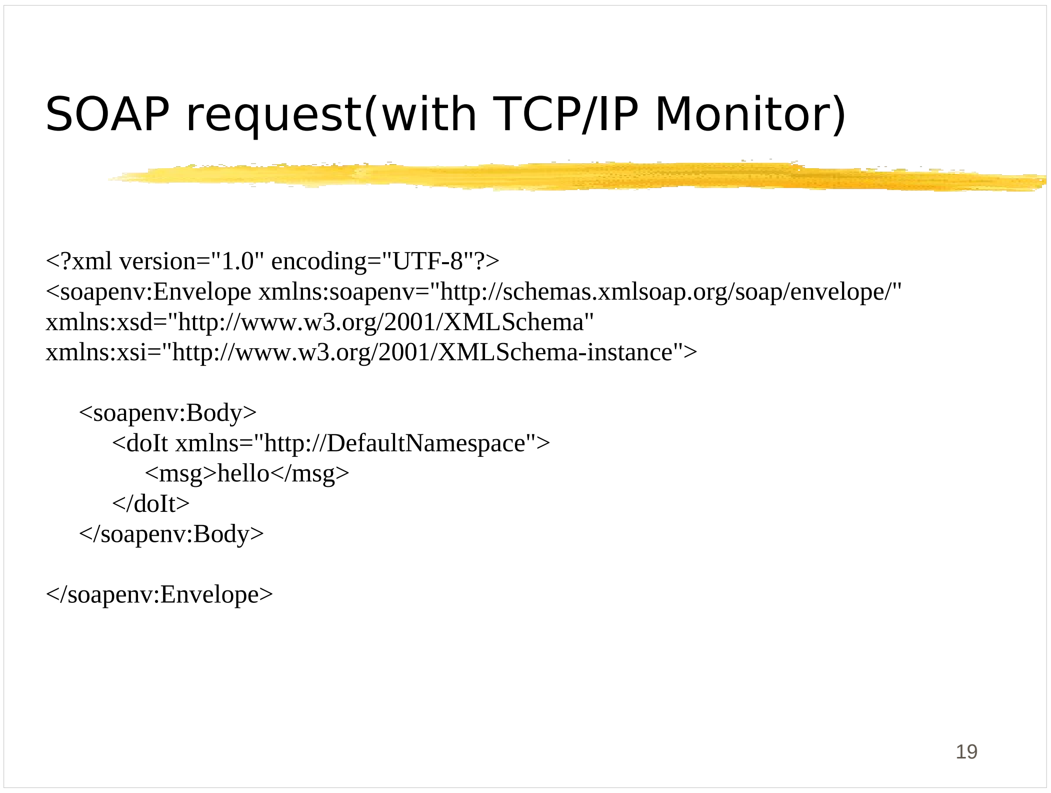


Generated WSDL

```
<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>
```

18

Very very verbose !

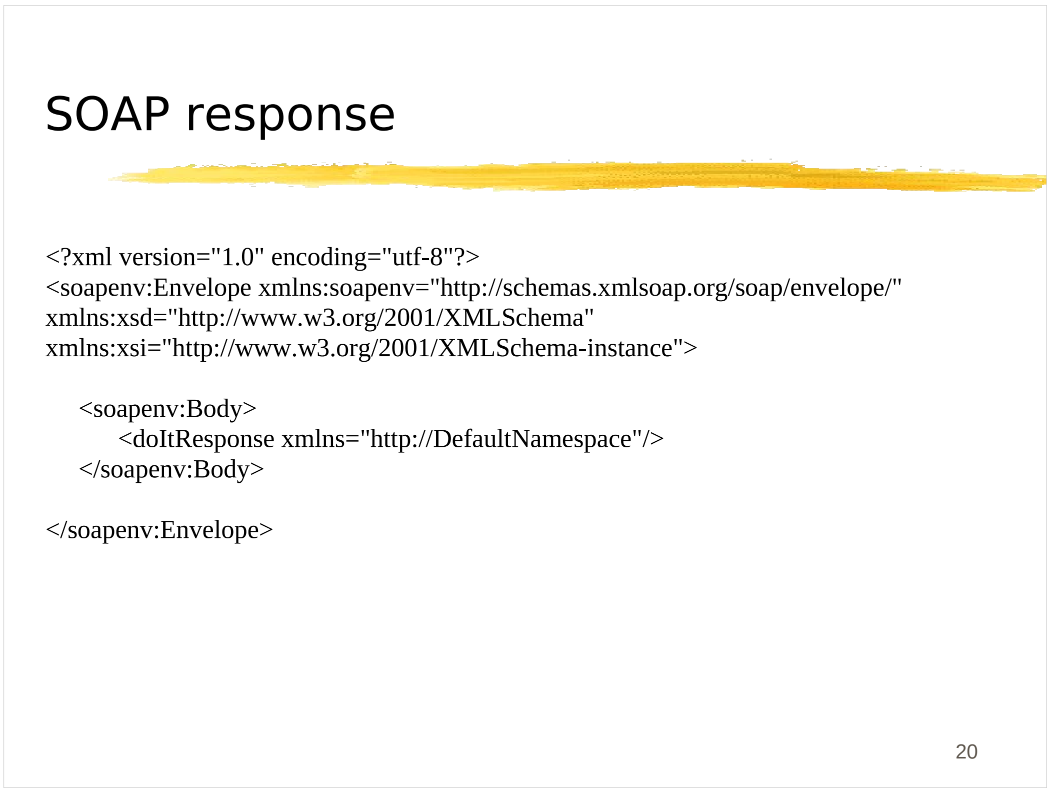


SOAP request(with TCP/IP Monitor)

```
<?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>
```

19

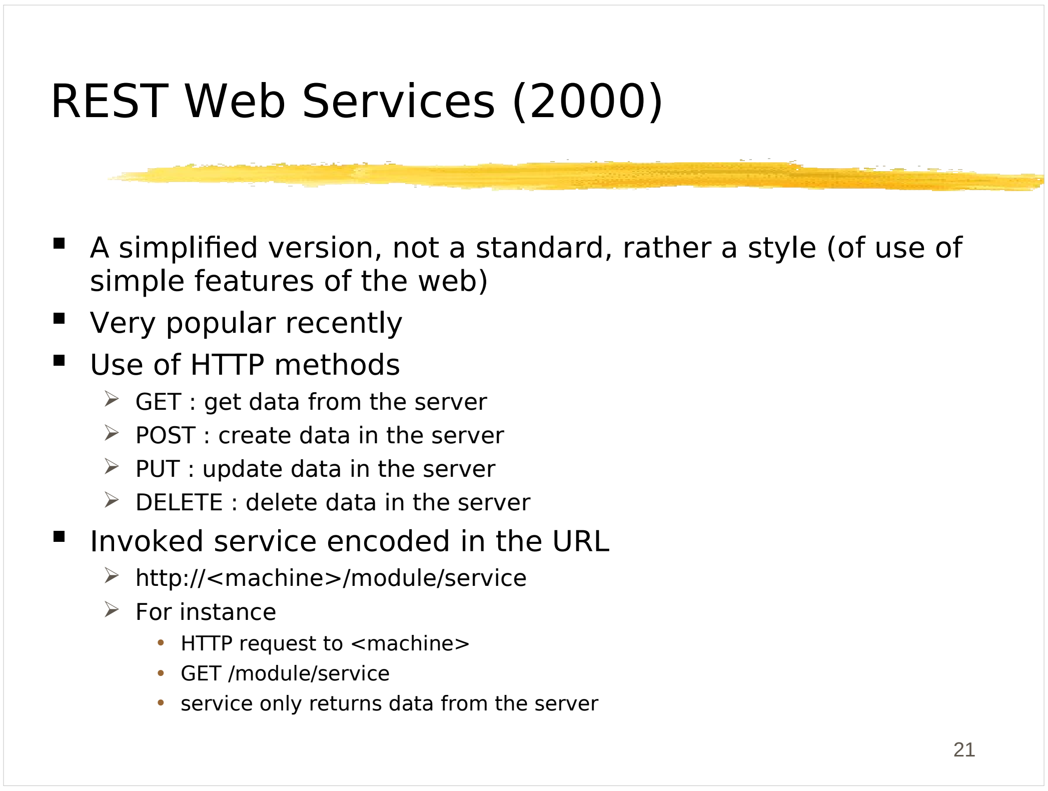
We can have a look at the SOAP request. This is simply a standardized format for exchanged messages.



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>
```

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)
- Very popular recently
- 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

SOAP/WSDL based WS were very popular few years ago. They are now obsolete.

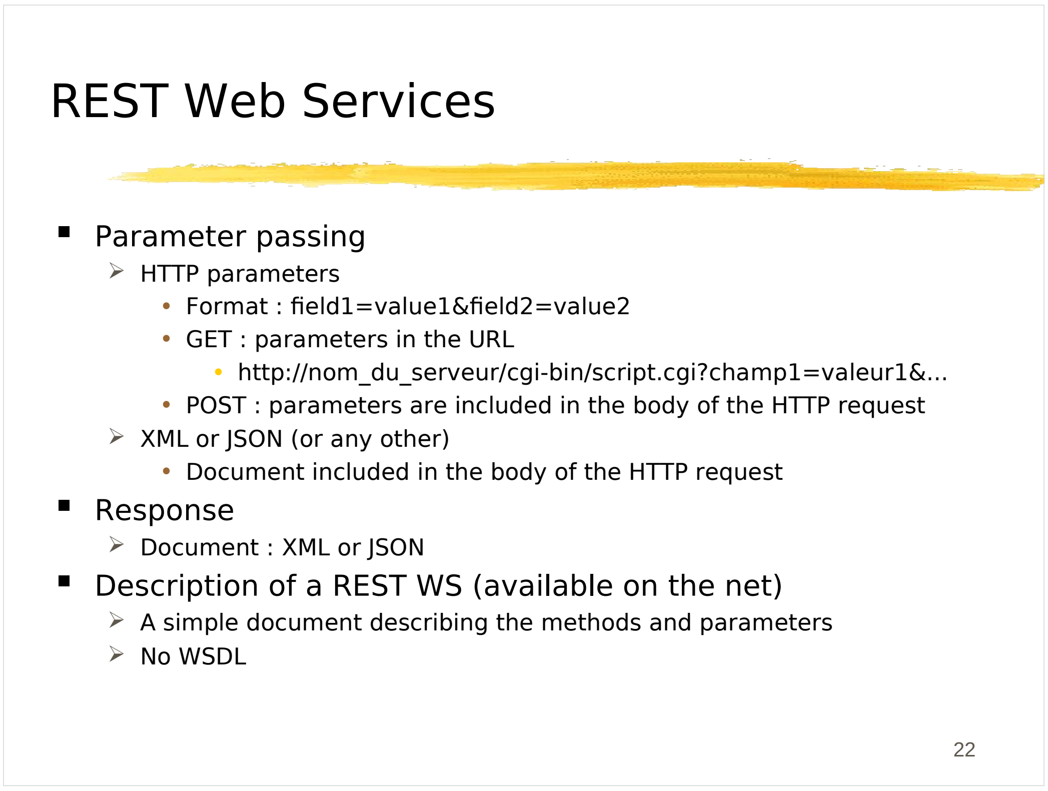
An evolution of WS is REST WS. This is a simplified version which is very popular now. Notice the REST WS is not a standard, but rather a recommendation or a style of implementation, based on simple features of the web.

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)
    - Document included in the body of the HTTP request
- Response
  - Document : XML or JSON
- Description of a REST WS (available on the net)
  - A simple document describing the methods and parameters
  - No WSDL

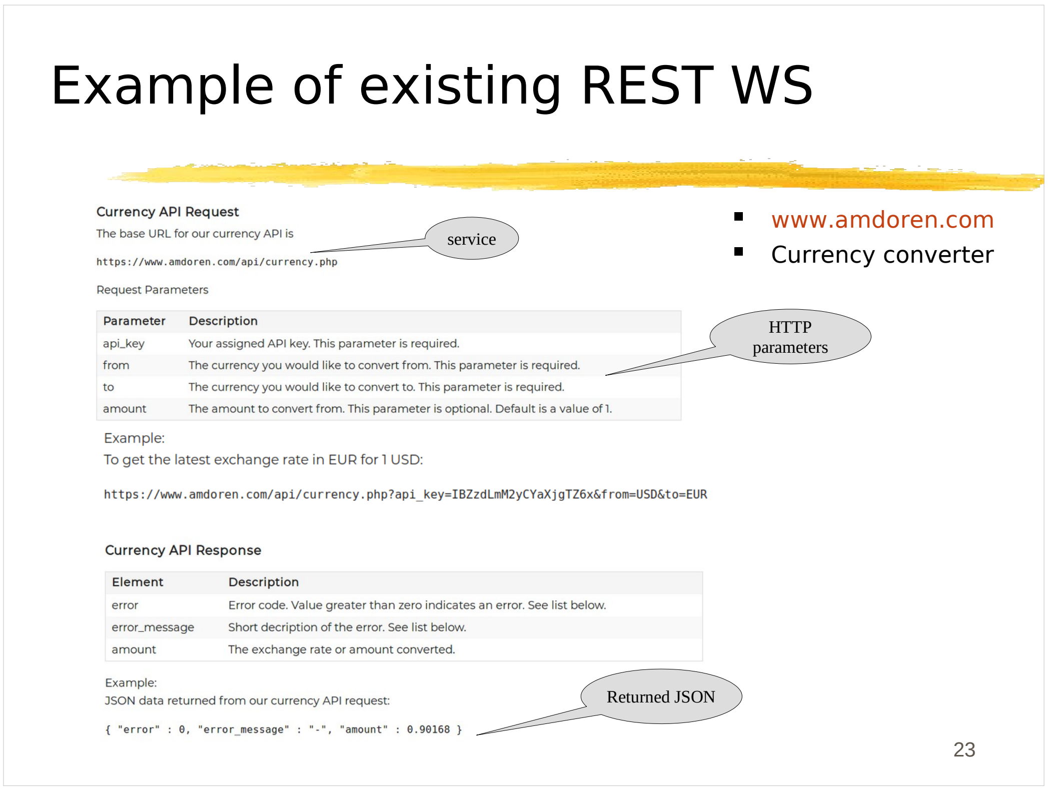
Parameter passing can be based on HTTP parameters. With HTTP parameters, parameters are encoded as a String field1=value1&field2=value2 …

This parameter sqtring 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 returns 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=IBZzdLmM2yCYaXjgTZ6x&from=USD&to=EUR
```

Currency API Response

Element	Description
error	Error code. Value greater than zero indicates an error. See list below.
error_message	Short decription 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 }
```

- 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 :

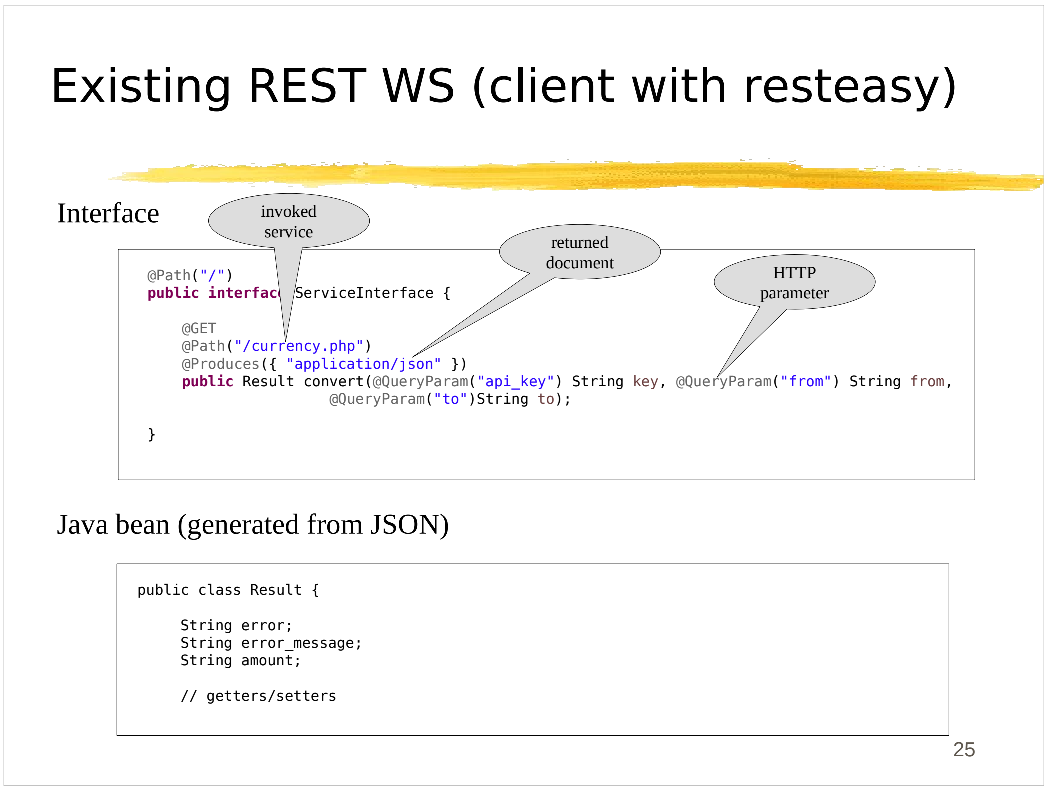
https://www.amdoren.com/api/currency.php

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

- 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 rest of the talk



Existing REST WS (client with resteasy)

Interface

invoked service
returned document
HTTP parameter

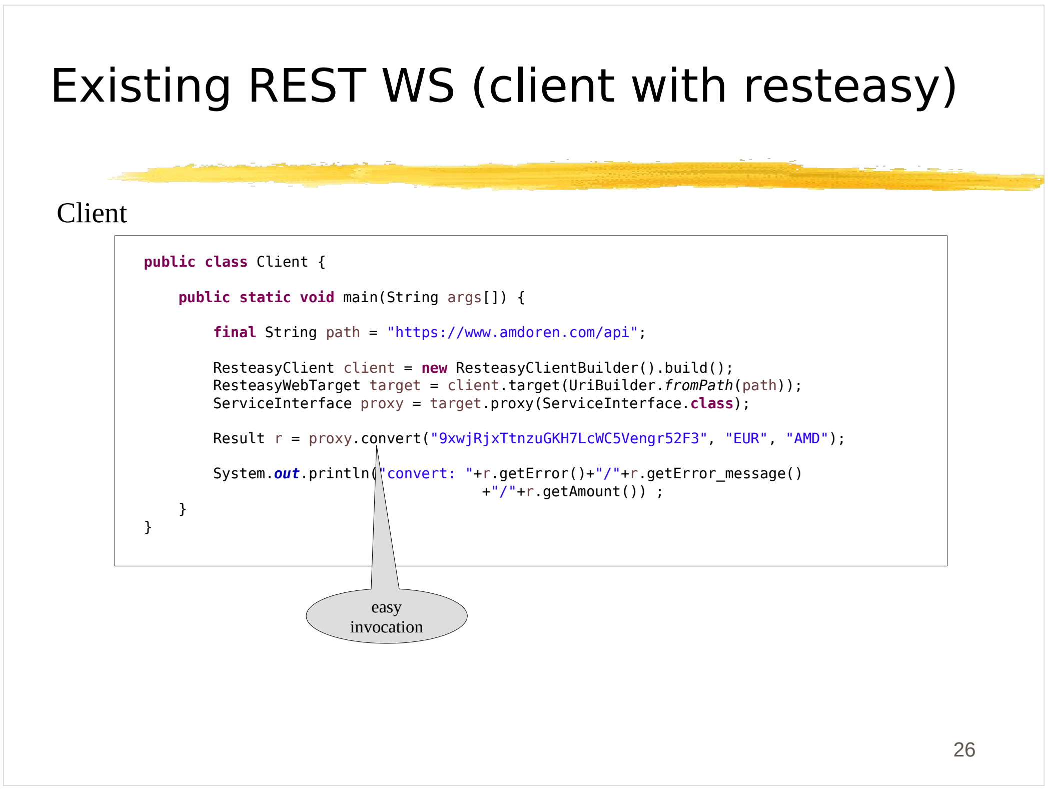
```
@Path("/")
public interface ServiceInterface {

    @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
```

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.

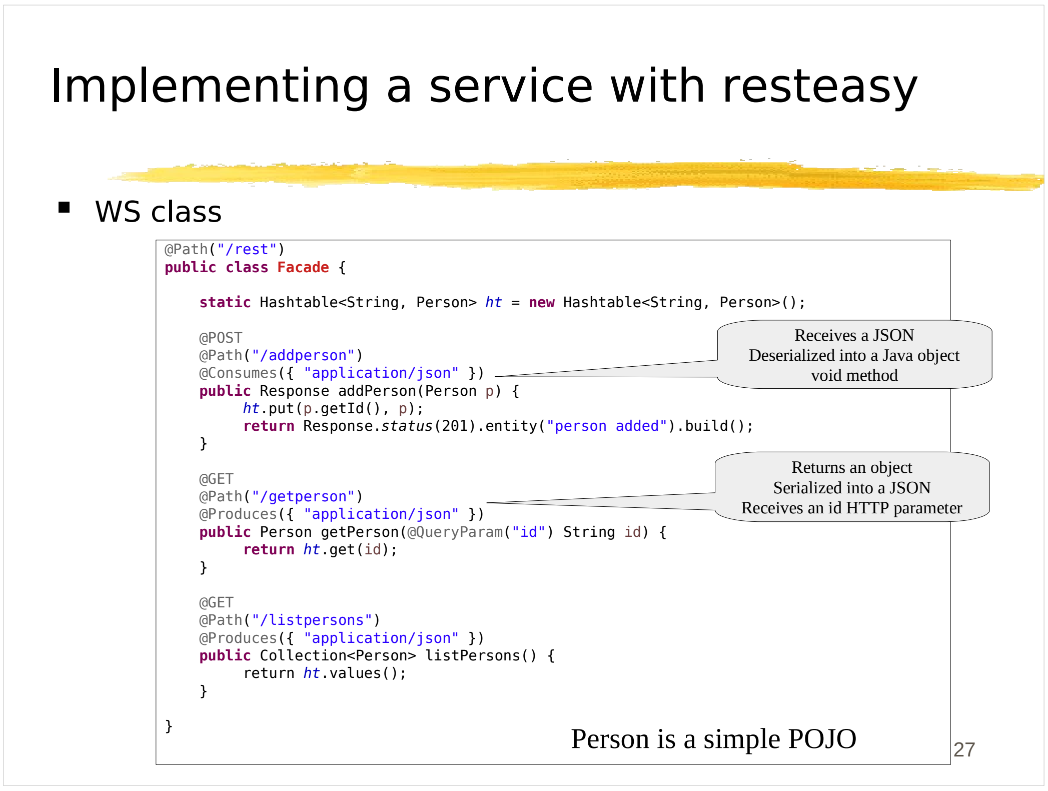


Existing REST WS (client with resteasy)

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()) ;
    }
}
```

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



Implementing a service with resteasy

- WS class

```
@Path("/rest")
public class Facade {

    static Hashtable<String, Person> ht = new Hashtable<String, Person>();

    @POST
    @Path("/addperson")
    @Consumes({ "application/json" })
    public Response addPerson(Person p) {
         ht.put(p.getId(), p);
         return Response.status(201).entity("person added").build();
    }

    @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

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

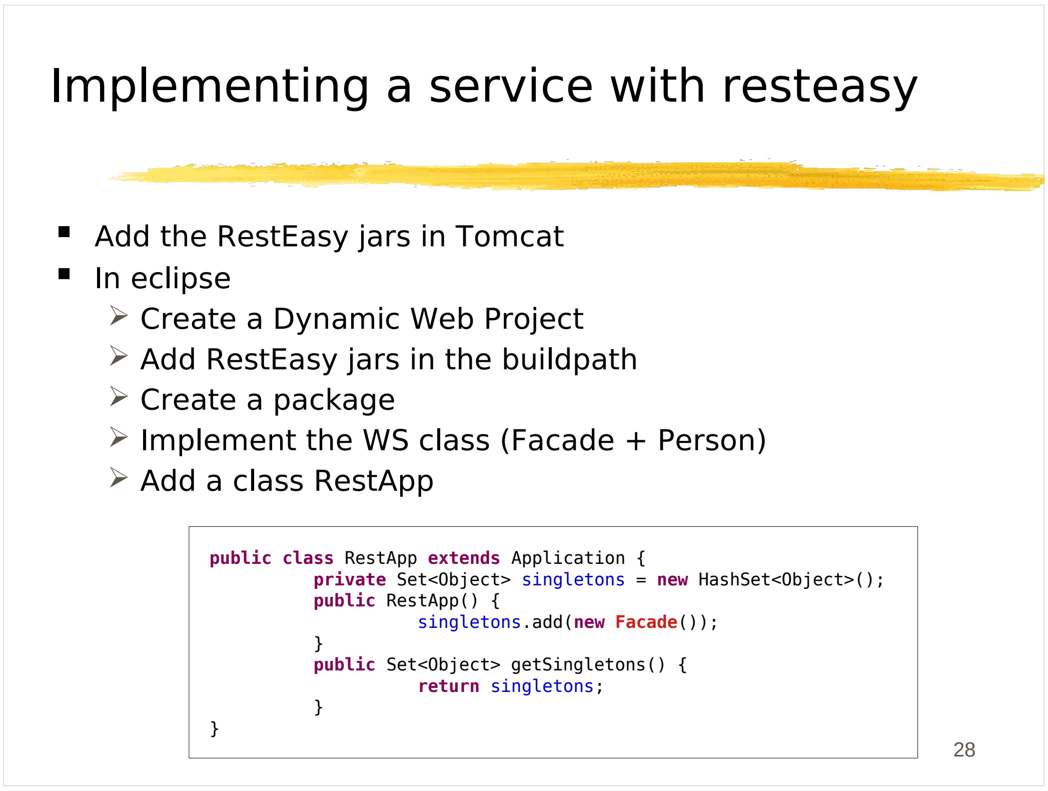
- @Path : specifies the element of the path associated with the class or the method. Here method addPerson() is associated with path /addperson

- @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

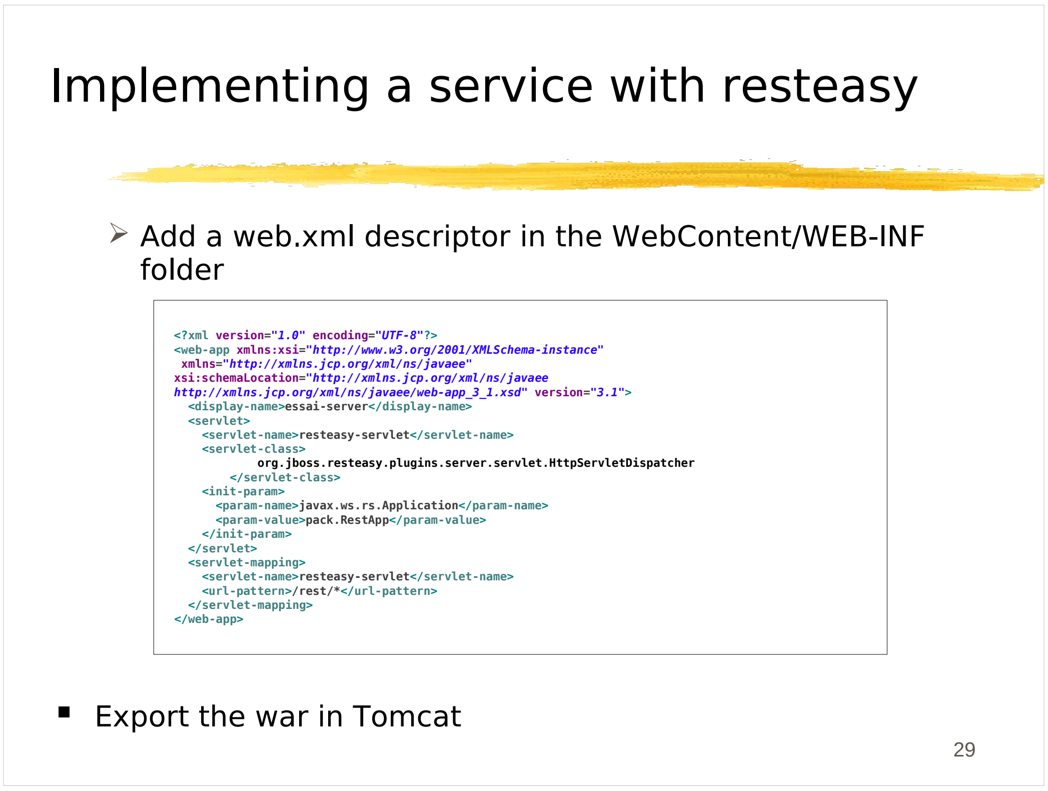
- Add the RestEasy jars in Tomcat
- In eclipse
  - Create a Dynamic Web Project
  - Add RestEasy jars in the buildpath
  - Create a package
  - Implement the WS class (Facade + Person)
  - Add a class RestApp

```
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)



Implementing a service with resteasy

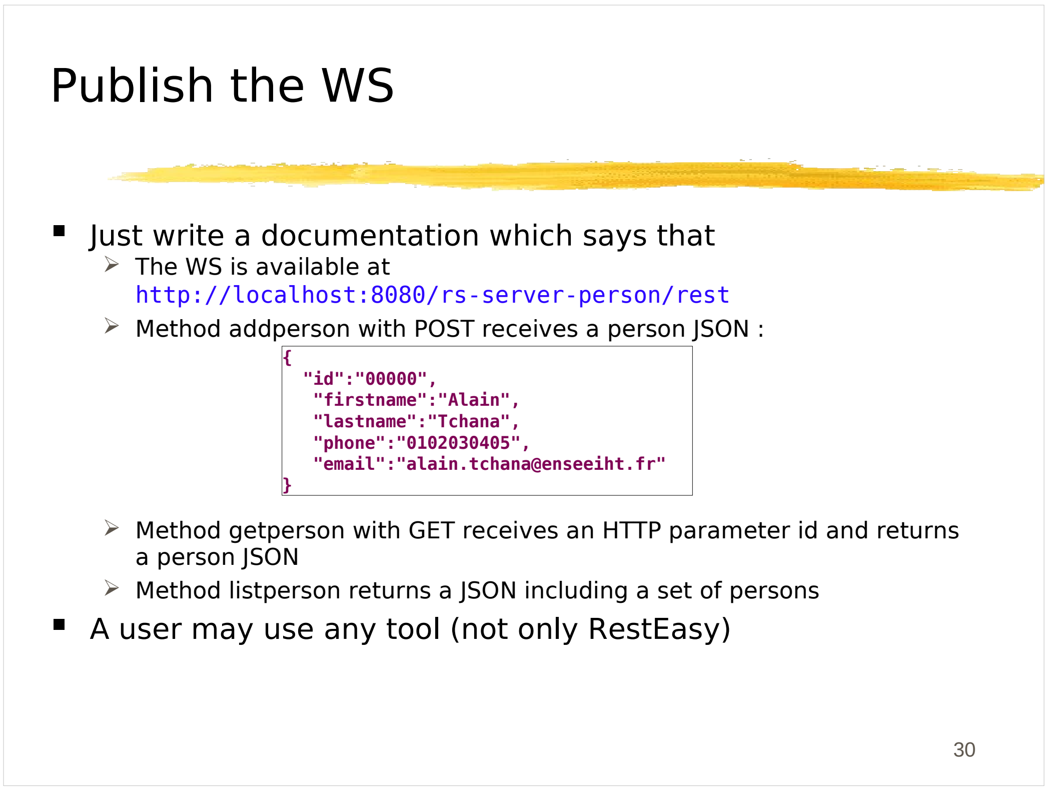
- Add a web.xml descriptor in the WebContent/WEB-INF folder

```
<?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>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

- Export the war in Tomcat

29

Add the descriptor and export a war



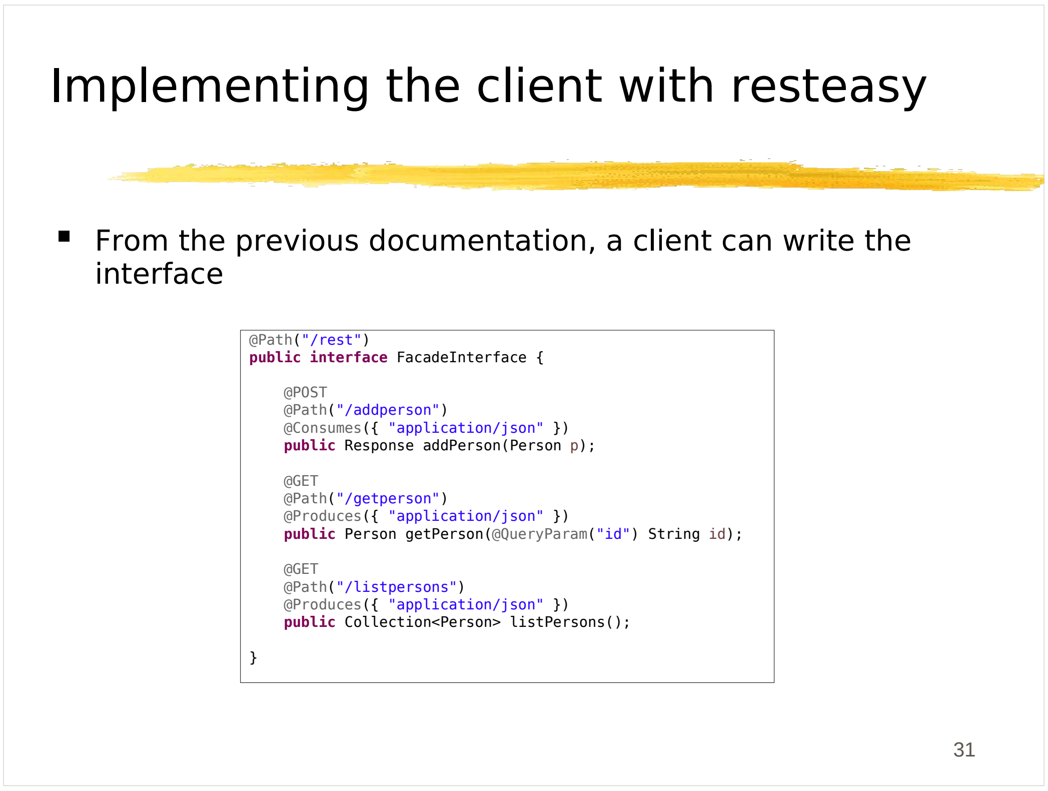
Publish the WS

- Just write a documentation which says that
  - The WS is available at http://localhost:8080/rs-server-person/rest
  - 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 user may use any tool (not only RestEasy)

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



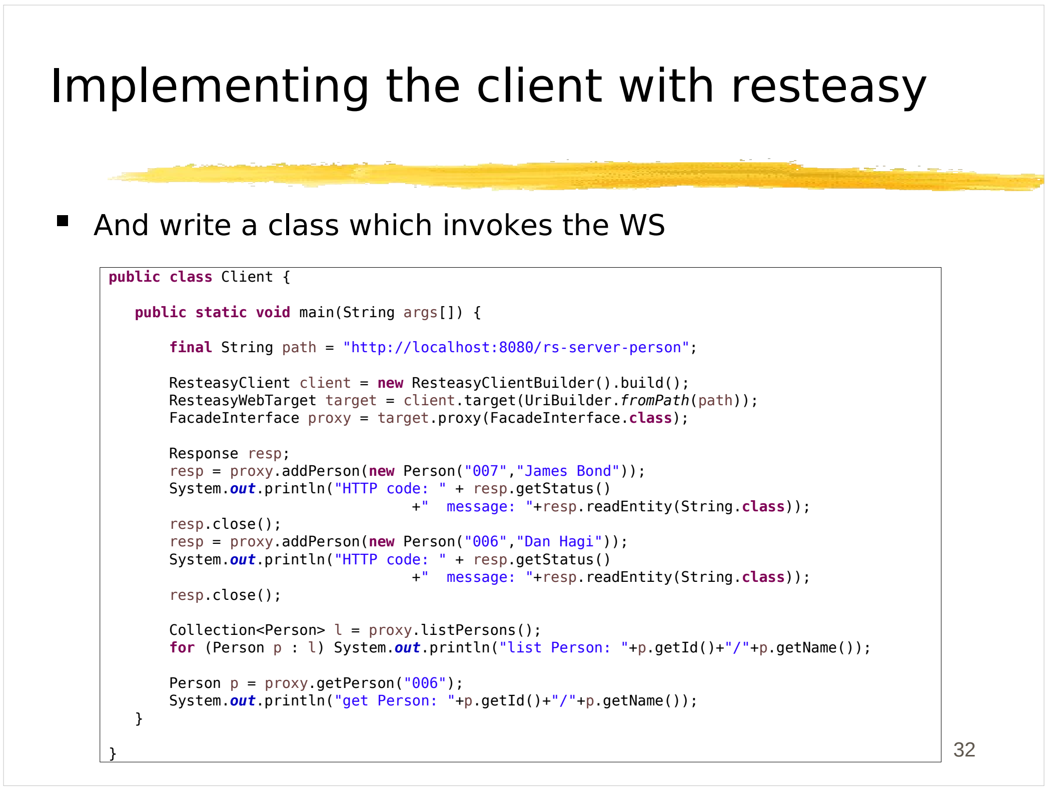
Implementing the client with resteasy

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

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

    @POST
    @Path("/addperson")
    @Consumes({ "application/json" })
    public Response 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();

}
```

31

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).



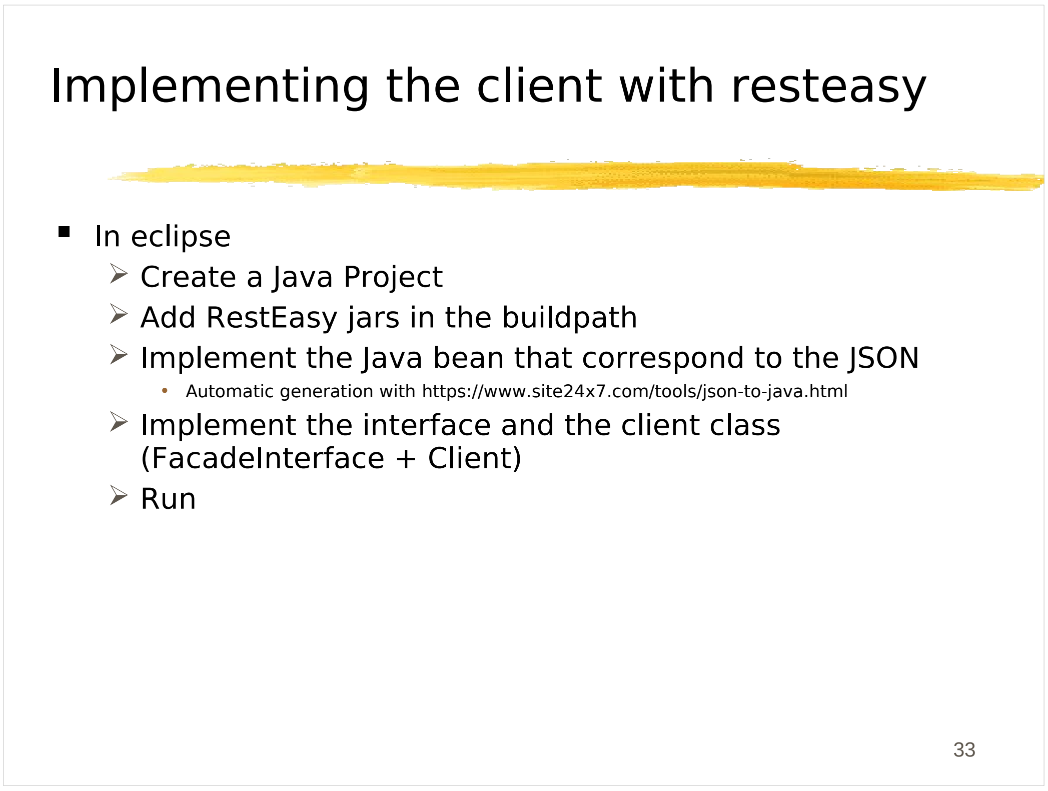
Implementing the client with resteasy

- And write a class which invokes the WS

```
public class Client {

   public static void main(String args[]) {

       final String path = "http://localhost:8080/rs-server-person";

       ResteasyClient client = new ResteasyClientBuilder().build();
       ResteasyWebTarget target = client.target(UriBuilder.fromPath(path));
       FacadeInterface proxy = target.proxy(FacadeInterface.class);

       Response resp;
       resp = proxy.addPerson(new Person("007","James Bond"));
       System.out.println("HTTP code: " + resp.getStatus()
                                   +"  message: "+resp.readEntity(String.class));
       resp.close();
       resp = proxy.addPerson(new Person("006","Dan Hagi"));
       System.out.println("HTTP code: " + resp.getStatus()
                                   +"  message: "+resp.readEntity(String.class));
       resp.close();

       Collection<Person> l = proxy.listPersons();
       for (Person p : l) System.out.println("list Person: "+p.getId()+"/"+p.getName());

       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.

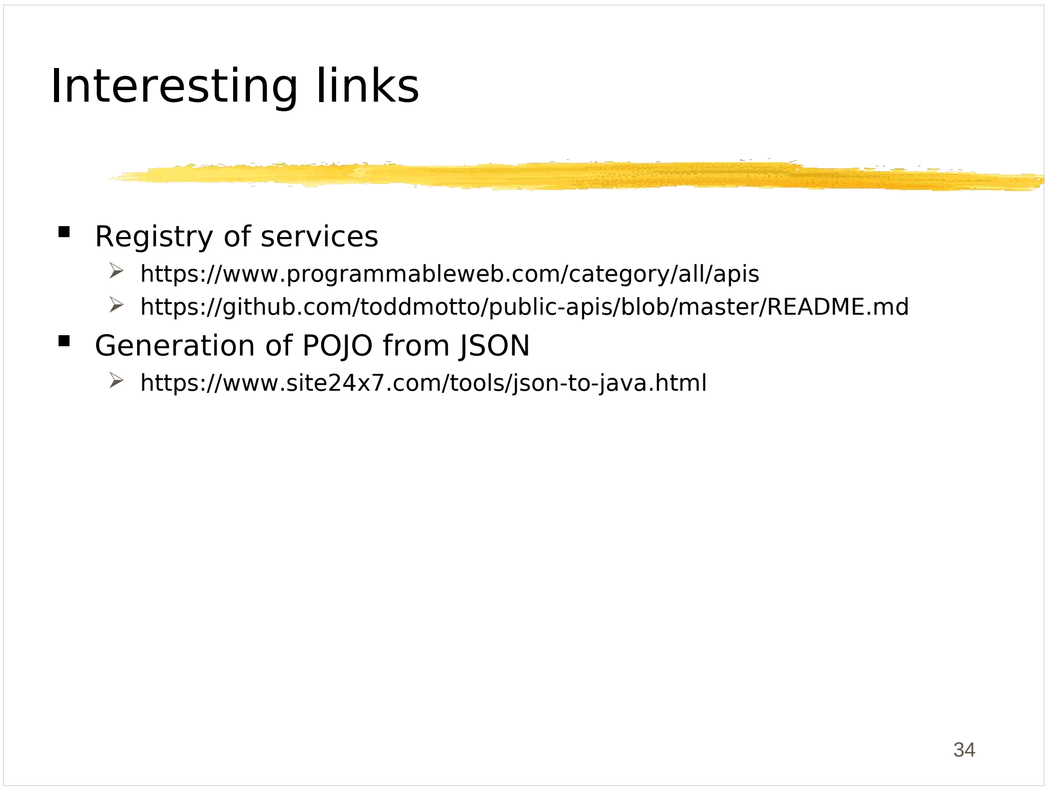
This proxy allows programming service invocations simply as method calls.



Implementing the client with resteasy

- In eclipse
  - Create a Java Project
  - Add RestEasy jars in the buildpath
  - 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 + Client)
  - Run

This is the procedure to run the client.



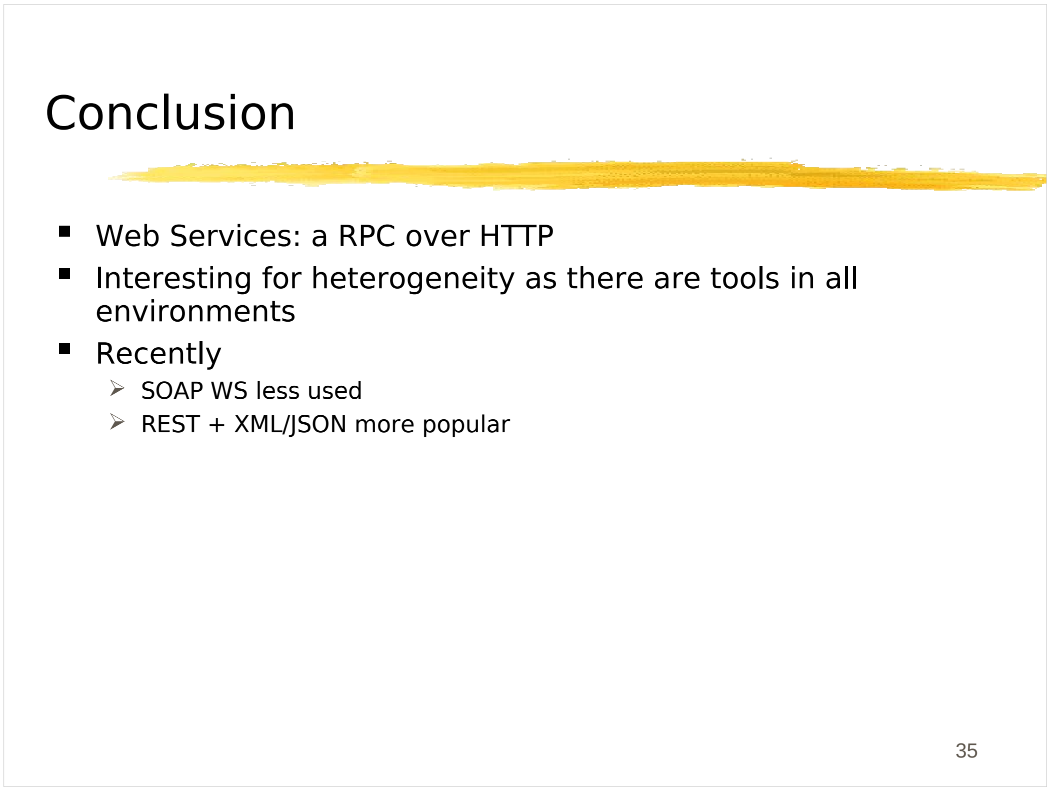
Interesting links

- Registry of services
  - https://www.programmableweb.com/category/all/apis
  - https://github.com/toddmotto/public-apis/blob/master/README.md
- Generation of POJO from JSON
  - https://www.site24x7.com/tools/json-to-java.html

Here are interesting links :

- sites where you can find interesting services

- a site which allows generating Java beans (POJO) from JSON



Conclusion

- Web Services: a RPC over HTTP
- Interesting for heterogeneity as there are tools in all environments
- Recently
  - SOAP WS less used
  - REST + XML/JSON more popular

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.