RESTful WebServices in Java

Chantal Taconet

September 2024

1 Introduction

1.1 REST API examples

- REST is a "URL friendly" way to retrieve distributed resources
- Well known examples of REST resources
  -  Google maps: API
    - ? where is this place `http://maps.googleapis.com/maps/api/geocode/json?latlng=40.714224,-73.961452` [1]
    - ? URL to get an address in Evry with GPS: lat=48.625595, lon=2.443234
  -  Open street map API
    - ? Where is this place `http://nominatim.openstreetmap.org/reverse?lat=48.858518&lon=2.294524&addressdetails=1`
  -  State of bike stations in Lyon, API
    - `https://api.jcdecaux.com/vls/v1/stations?contract=lyon&apiKey=91f170cdabb4c3227116c3e871a63e8d3ad148ee`

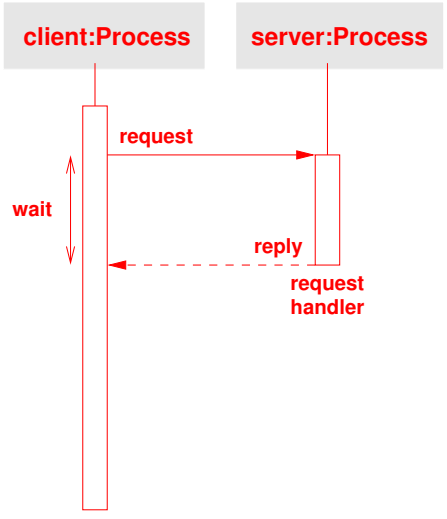
1. Access restricted you need a google account and a key

2 REST architectural style

2.1 REST: Representational State Transfer

- **Architectural style** defined by *Roy Fielding* in a PhD [Fielding, 2000]
- Described by **six** identified **constraints**
- World Wide Web conforms to the REST architectural style
- Components that conform to this architectural style are called **RESTful Web services**

2.2 Constraint 1: Client-server architecture



Synchronous call

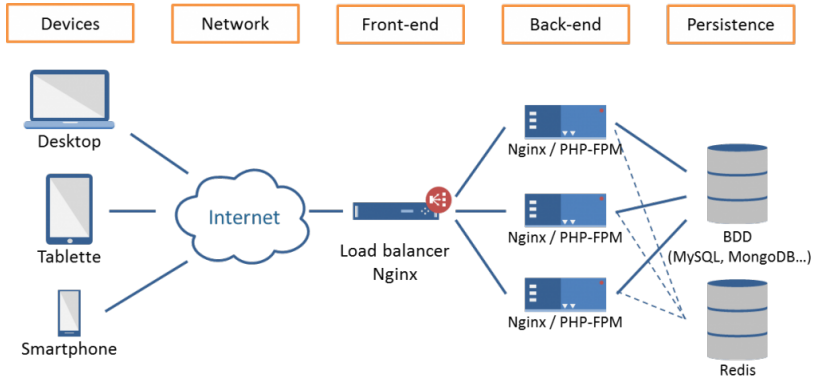
- Allows the components (**client** and **server**) to evolve independently: the link is the API (Application Programming Interface)
- Separate the interface (API) concerns from the data storage concerns (implementation of the **resource**)
- Supported by multiple platforms and multiple languages

2.3 Constraint 2: Stateless

From Roy Fielding dissertation

- Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.
- Session state is therefore kept entirely on the client.

Advantage of the stateless constraint



- **Scalability**: as each request may be handled by a different server, the number of servers may be augmented as necessary

2.4 REST: Architectural style : 6 constraints

- **Client/server** architecture
- **Stateless**: no client context on the server
- **Cacheable**: clients can cache responses
- **Layered system**: clients and servers may be connected through intermediate layers (e.g. proxies)
- **Code on demand**: the state may include code (e.g. javascript)
- **Uniform interface** between clients and servers

2.5 Uniform interface: CRUD operations

- Requests and responses are built around the transfer of representations of resources
- Requests are one of the **four CRUD Operations**:
  - **Create** resource $\mapsto$ **POST** http method
  - **Read** resource $\mapsto$ **GET** http method
  - **Update** resource $\mapsto$ **PUT** http method
  - **Delete** resource $\mapsto$ **DELETE** http method

2.6 Are these operations sufficient to build an application?

Resource	Create POST	Read GET	Update PUT	Delete DELETE
Collection	Create entry	List entries	Replace collection	Delete collection
Element	/	Get element	Replace element	Delete element

2.7 REST resource

- Any (Web) resource
- Identified by a **global identifier** (e.g. URI [Uniform Resource Identification])
- **State** of a resource may be transfered through a **representation** of this resource

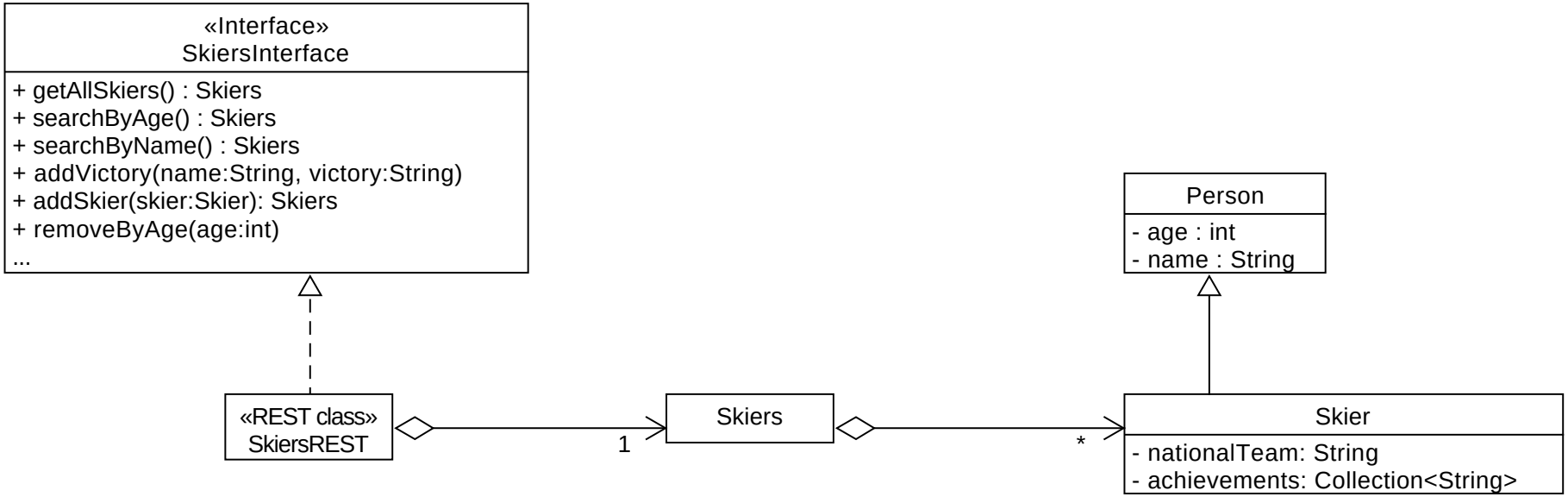
2.8 URI Naming conventions

- Collection of resources: e.g., /skiers
- Single resource: e.g., /skiers/{skierid}
  - {skierid} is a **parameter path**
- Subcollection: e.g., /skiers/{skierid}/achievements
- Single resource: e.g., /skiers/{skierid}/achievements/{achievementId}
- Controller: e.g., /skiers/{skierid}/change-name/{new-name}
- Find: /skiers?age=41
  - age is a **query parameter**


When resources are named well: an API is intuitive and easy to use


If done poorly, that same API can feel difficult to use and to understand

2.9 Skiers example



2.10 HATEOAS

Hypermedia as the Engine of Application State

? What is it: including hypermedia links into a resource state

Objective

- A client of a REST application need only to know a single fixed URL
- Related resources should be discoverable dynamically from that URL

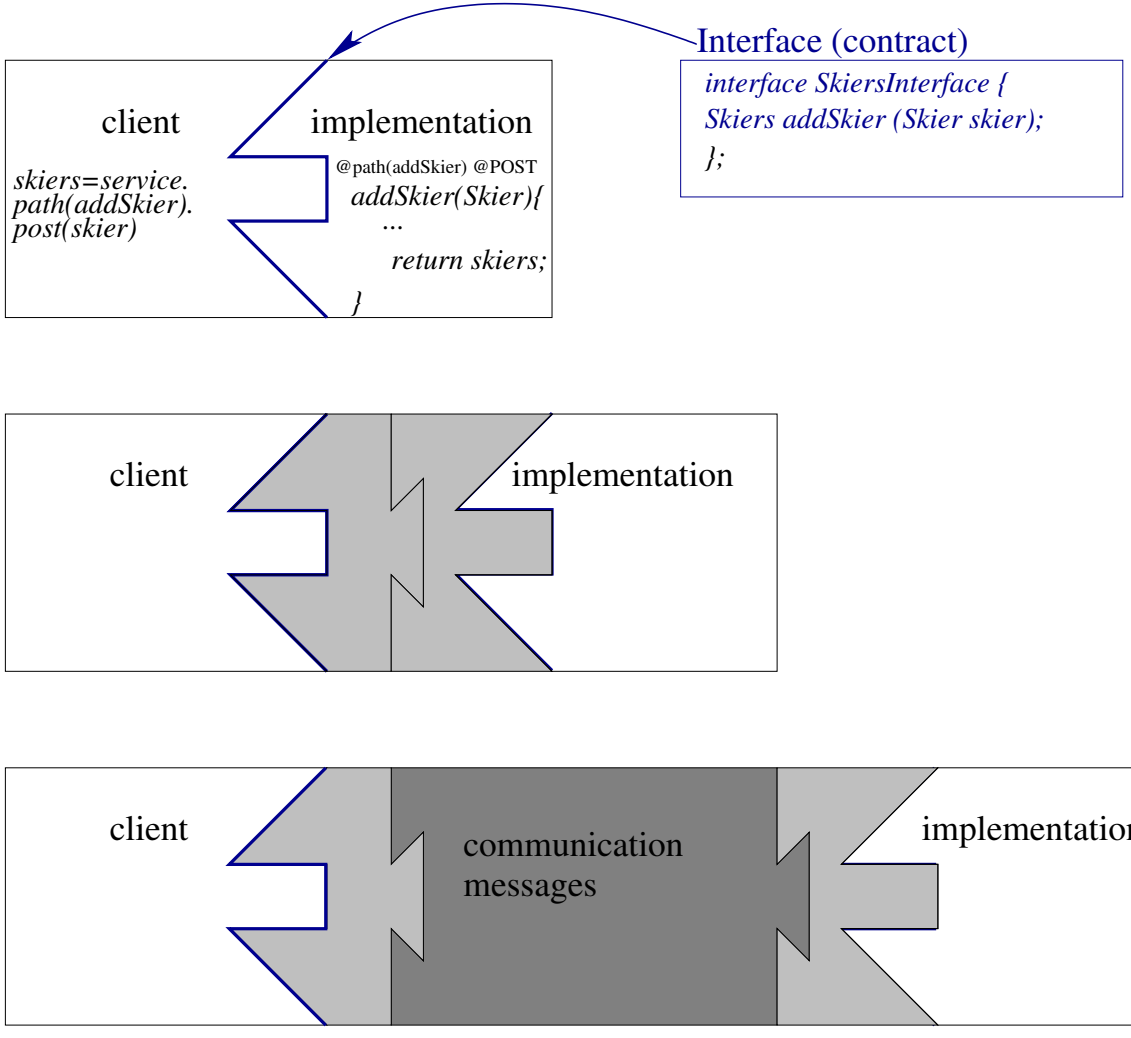
HOW: Hyperlinks included in the representations of returned resources

JSON EXAMPLE

```
{
  "person": {"name": "Kelly"},
  "nationalTeam": {
      "Norway",
          "_links": {
               "nbskiers": {"href":"http://rest.norway-ski-team.no/nbskiers"}
   },
  "achievements":["12 Olympic Medals",
                  "9 World Championships",
                 ]
}
```

3 Marshalling/unmarshalling

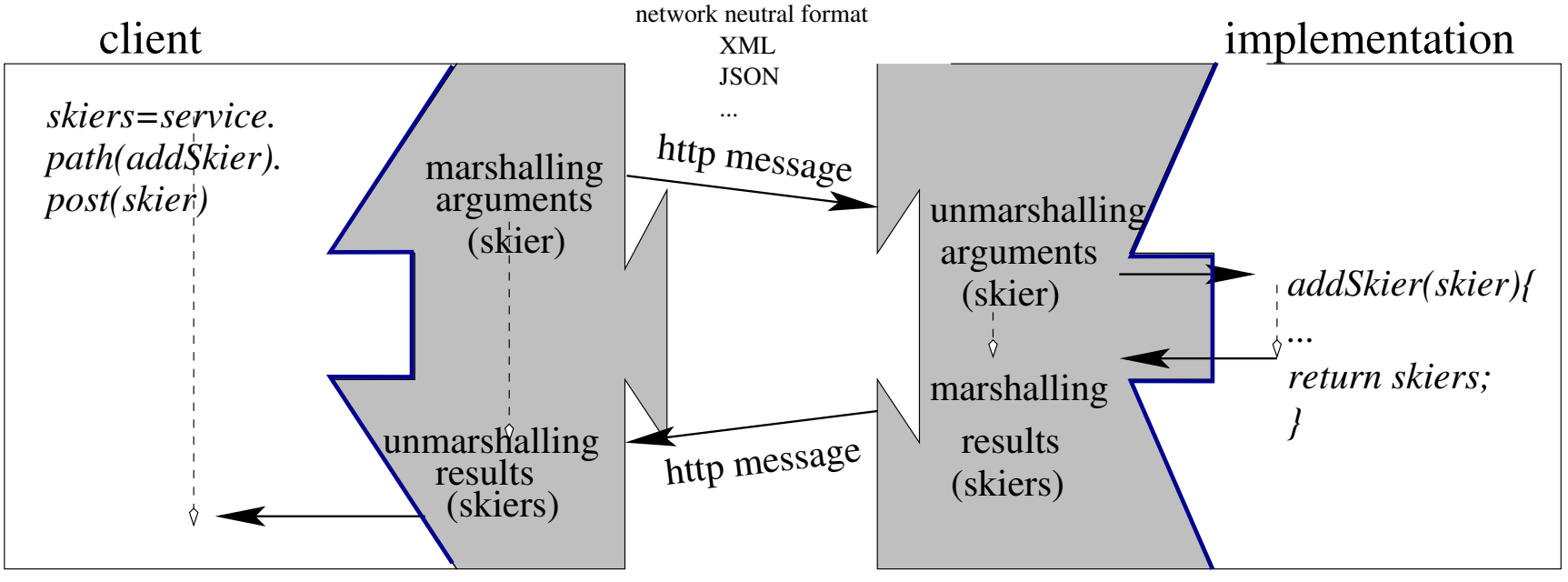
3.1 From resource, to remote resource



3.2 Marshalling and unmarshalling

Exemple **Client java**, **serveur php**, représentation en **JSON**

- **Marshalling** : e.g. **Java instance** $\mapsto$ representation JSON
- **Unmarshalling** : e.g. one representation JSON $\mapsto$ **Php instance**



3.3 Representation formats

- Resources are distinct from their possible representations
- Format of a representation (i.e. **content type**) is defined by an Internet media type (previously known as a MIME type)
- Some common formats
  - plain text: *text/plain*
  - html: *text/html*
  - xml: *text/xml*, *application/xml*
  - code: *application/javascript*
  - json: *application/json*
  - image: *image/jpeg*, *image/png*, *image/**
  - video: *video/mpeg*

4Hyper Text Transfer Protocol: basics reminder

4.1 HTTP GET Request message

```
1 GET /hello HTTP/1.1
2 Accept: text/plain, text/html
3    %———empty line: end of header
```

- Sent to a web server to access one of its web resource
  - Request message (message method, identification of the resource inside the server, HTTP version)
    - For instance: `GET /hello HTTP/1.1`
  - Request Headers
    - accepted content types (e.g. `Accept: text/plain, text/html`)
    - accepted charsets (e.g. `Accept-Charset: utf-8`)
    - cookie (e.g. Cookie: `Version=1; Skin=new;`)
  - Request body (empty for a get)

4.2 HTTP GET Response message

```
1  HTTP/1.1 200 OK     return code
2  Date: Mon, 11 Nov 2013 17:47:24 GMT  header (begin)
3  Server: Apachee/2.2.3 (Debian GNU/Linux)
4                  Perl/v5.8.4 PHP/5.2.6
5  Last-Modified: Wed, 28 Apr 2012 15:55:02 GMT
6  Content-length: 327
7  Content-type: text/html
8                                              empty line (end of header)
9  <HTML>     content
10 ... document HTML
11 </HTML>
```

- Return code (line 1)
  - 100 - 199: Information message
  - 200 - 299: Success (e.g., 200 OK)
  - 300 - 399: Redirections
  - 400 - 499 : client-side errors (e.g., 404 Not Found, 403 Forbidden)
  - 500 - 599 : server-side errors (e.g., 500 Internal Server Error)
- Header (line 2–7)
- Resource content (line 9-11)

4.3 HTTP GET give it a try

Give it a try

1. Visualize this simple page on your favourite navigator `http://checkip.dyndns.org/` and vizualize the headers with the network inspector of your navigator
2. Visualize the result with the *curl* command

```
curl http://checkip.dyndns.org/
```

3. Connect to the web server with the telnet command

```
telnet checkip.dyndns.org 80 % establish a connexion
GET /
  % empty line
```

4. Use a REST client (such as postman)

5 From a Java instance to a neutral representation

5.1 Java instance to State representation I

- Several marshalling/unmarshalling means
  - Java serialization : binary representation

```
1 class MyClass implements Serializable {}
2
3 instance =new MyClass();
4 final FileOutputStream fichier = new FileOutputStream("file.ser");
5 ObjectOutputStream oos = new ObjectOutputStream(fichier);
6 oos.writeObject(instance);
```

  - JAXB : XML Document
  - Json : JavaScript Object Notation

5.1 Serialization

- Automatic serialization concerns
  - Loop: Object graph with cycles
  - Multiple references: Object graph with multiple reference paths to the same object

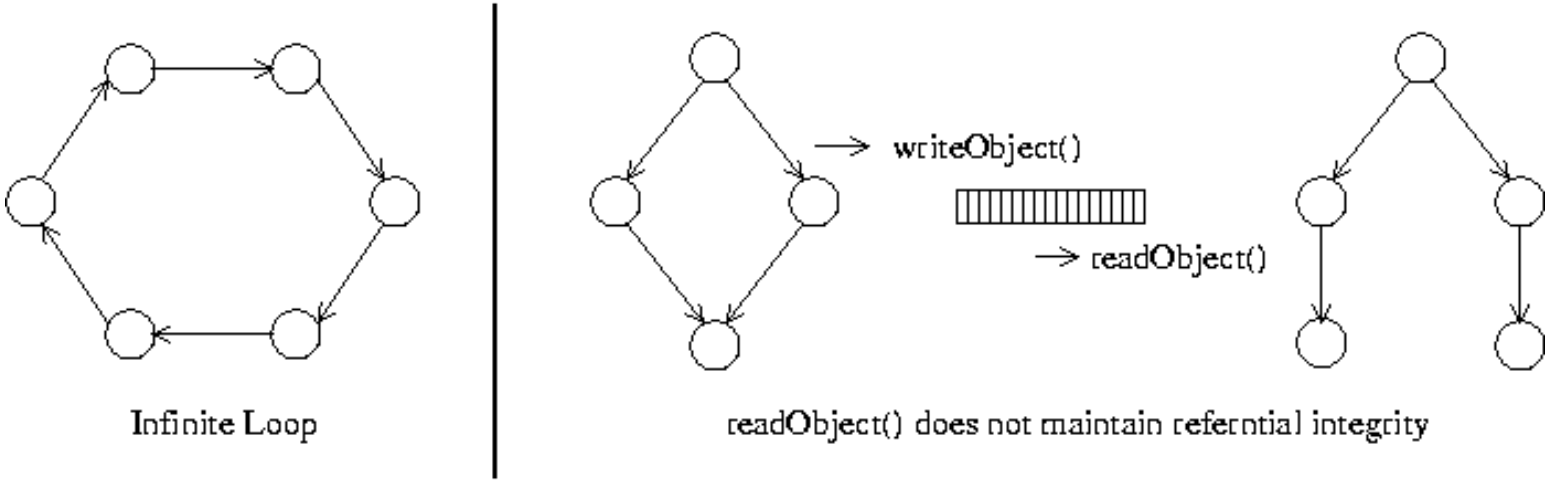


Figure source: Javadoc DataSerialize

5.2 Json (Javascript Object Notation)

- "*JSON is a **lightweight data-interchange format**. It is easy for humans to read and write. It is easy for machines to parse and generate.*" (json.org)
- Native representation of object in JavaScript
- Many programming languages include code to generate and parse JSON-format data

5.2 From Java instance to Json document

java Class

```java
public class Person {
    private String name;
    private int age;
    private String gender;
}
```

Json Schema

```json
{
  "type":"object",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "integer"
    },
    "gender": {
      "type": "string"
    }
  }
}
```

Java object

```java
Person p=new Person("Bjoern Daehlie", 41, \
      "Male");
```

Json document

```json
{
  "name": "Bjoern Daehlie",
  "age": 41,
  "gender": "Male"
}
```

5.2 Marshall a Java instance in Json

- Many java libraries to serialize/deserialize Json strings, **jakarta** is one of them
- Skier Example in ExamplesREST/REST-JSON-in-jakarta
- A skier in Json

```
skier ={
  "nationalTeam":"Norway",
  "achievements":["12 Olympic Medals",
                  "9 World Championships",
                  "Winningest Winter Olympian",
                  "Greatest Nordic Skier"
                 ],
  "name":"Bjoern Daehlie",
  "age":41,
  "gender":"Male"
}
```

5.2 Skier in json

Marshall.java[2]

```
1  import javax.json.bind.Jsonb;
2  import javax.json.bind.JsonbBuilder;
3   . . .
4          Skier skier = createSkier();
5
6          Jsonb jsonb = JsonbBuilder.create(); //Create a Json builder
7
8          String json = jsonb.toJson(skier); // Marshall from java instance to json
9          Skier clone = (Skier) jsonb.fromJson(json, Skier.class); // Unmarshall from json to java \
               instance
10
11         Assert.assertEquals("clone equals skier", clone, skier); // Verify
```

2. Example from *REST examples: directory REST-JSON-in-jakarta class Marshall*

5.3 JAXB — Java Architecture for XML Binding

- JAXB used to transfer complex java objects in XML structured strings
  - Marshalling: Convert a Java object into an XML document
  - Unmarshalling: Convert an XML document into a Java Object

5.3 JAXB primitive data types

- Java basic types have a representation in xs types

Java type	xs type
java.lang.String	xs:string
int	xs:int
double	xs:double
boolean	xs:boolean
java.util.Date	xs:dateTime

- ? What about complex type ?

5.3 From Java instance to XML document

java Class

```
1 public class Person {
2     private String name;
3     private int age;
4     private String gender;
5 }
```

XSD schema

```
1 <xs:schema version="1.0"...>
2  <xs:complexType name="person">
3   <xs:sequence>
4    <xs:element name="age" type="xs:int"/>
5    <xs:element name="gender" type="xs:string" \
           minOccurs="0"/>
6    <xs:element name="name" type="xs:string" \
           minOccurs="0"/>
7  </xs:sequence>
8 </xs:complexType>
9 </xs:schema>
```

Java object

```
1 Person p=new Person("Bjoern Daehlie", 41, \
       "Male");
```

XML document

```
1 <person>
2     <name>Bjoern Daehlie</name>
3     <age>41</age>
4     <gender>Male</gender>
5 </person>
```

5.3 JAXB annotations I

Annotation	Description
@XmlRootElement(namespace = "namespace")	Root element for an XML tree
@XmlType(propOrder = "field2", "field1",..)	XSD Type, order of fields
@XmlAttribute	Translated into an attribute (rather than an element)
@XmlTransient	Not translated into XML
@XmlAccessorType(XmlAccessType.FIELD)	All attributes translated (by default, only public + getter/setter)
@XmlElementWrapper(name="")	Add a wrapper XML element
@XmlElement(name = "newName")	Rename a field (element)

5.3 Skier example

The JAXB examples are in the directory *REST-JAXB-01*
Annotations for the Skier class

```
1  import javax.xml.bind.annotation.*;
2
3  @XmlRootElement // XML Root
4  @XmlAccessorType(XmlAccessType.FIELD) // All the fields, even the private ones are marshalled in XML
5  public class Skier extends Person {
6      private String nationalTeam;
7      @XmlElementWrapper(name = "achievements") // Addition of a wrapper for the collection
8      @XmlElement(name = "achievement") // Name of the elements in the collection
9      private Collection<String> achievements;
10
11     public Skier() {}
12     public Skier(final Person person, final String nationalTeam, final Collection<String> \
           achievements) {
13         super(person);
14         this.nationalTeam = nationalTeam;
15         this.achievements = achievements;
16     }
17 }
```

5.3 Skier example, Person, parent class

Person.java[3] (not a root document)

```
1
2  import javax.xml.bind.annotation.*;
3
4  @XmlAccessorType(XmlAccessType.FIELD) // All the fields, even the private ones are marshalled in XML
5  public class Person {
6      private String name;
7      private int age;
8      private String gender;
9
10     public Person() { }
11     public Person(final Person person) {
12         this(person.name, person.age, person.gender);
13     }
14     public Person(final String name, final int age, final String gender) {
15         this.name = name;
16         this.age = age;
17         this.gender = gender;
18     }
19 }
```

3. REST examples: directory REST-JAXB-01

5.3 Skier example, XML root document

Example XML Document for a Skier object[4]

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<skier>
    <name>Bjoern Daehlie</name>
    <age>41</age>
    <gender>Male</gender>
    <nationalTeam>Norway</nationalTeam>
    <achievements>
        <achievement>12 Olympic Medals</achievement>
        <achievement>9 World Championships</achievement>
        <achievement>Winningest Winter Olympian</achievement>
        <achievement>Greatest Nordic Skier</achievement>
    </achievements>
</skier>
```

4. *REST examples: directory REST-JAXB-01*

5.3.1 JAXB in action

-  Using the JAXB API to marshall and unmarshall; Marshall.java[5]

```
1  import javax.xml.bind.*;
2
3      Skier skier = createSkier();
4
5      // Create a Marshaller for the skier class
6      JAXBContext ctx = JAXBContext.newInstance(Skier.class);
7      Marshaller m = ctx.createMarshaller();
8      m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);
9
10         // Marshal and Write on a file
11     FileOutputStream out = new FileOutputStream(FILE_NAME);
12     m.marshal(skier, out);
13     out.close();
14
15     // Read from the file and Un-marshal
16     Unmarshaller u = ctx.createUnmarshaller();
17     Skier clone = (Skier) u.unmarshal(new File(FILE_NAME));
18
19     Assert.assertEquals("\n clone equals skier", clone, skier);
```

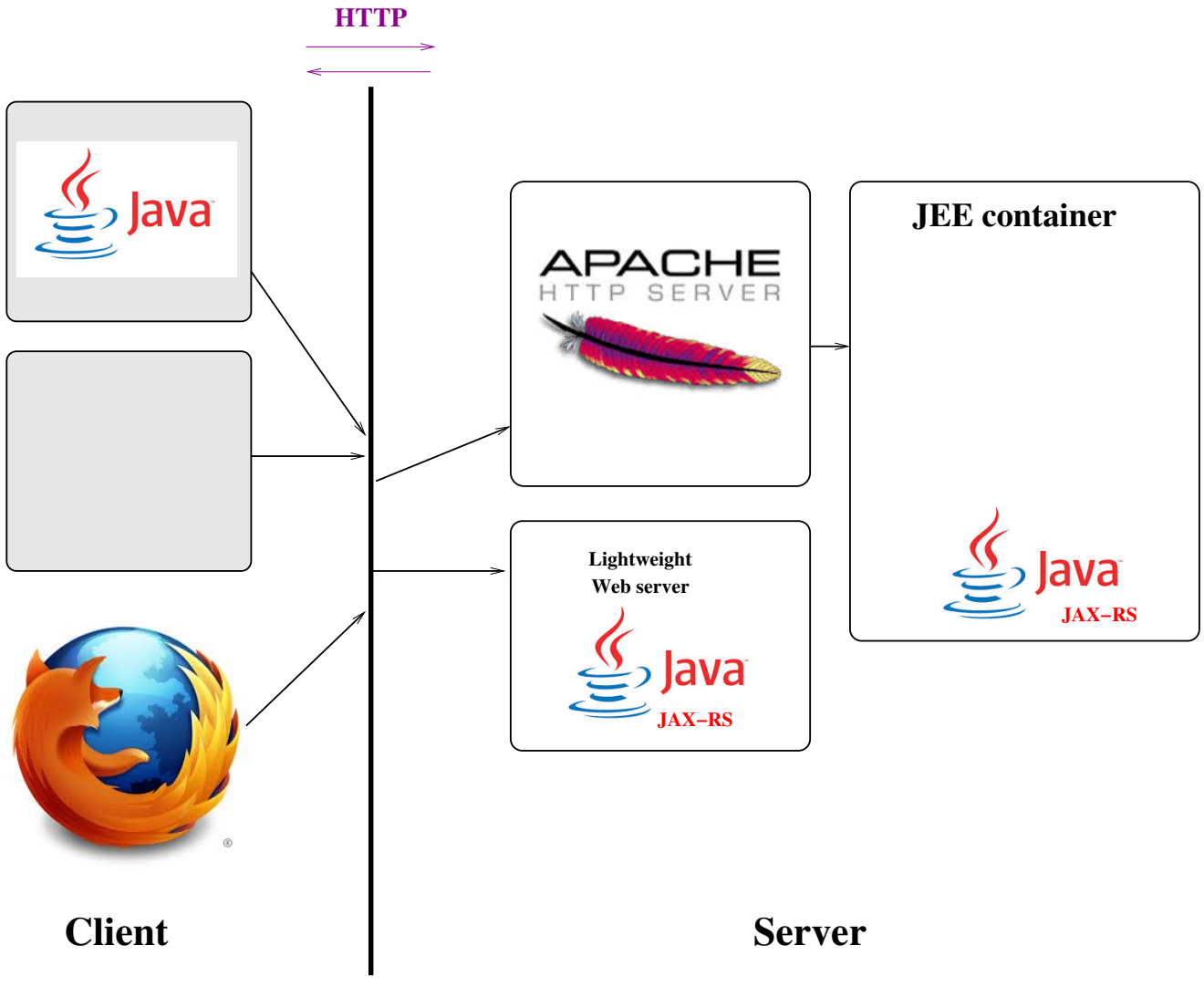
5. REST examples: directory REST-JAXB-01

5.3 Handling specific marshalling/unmarshalling JAXB I

- @XmlTransient attribute not marshalled
- *beforeMarshal* and *afterMarshal*: callbacks called (when defined) before and after marshalling
- *beforeUnmarshal* and *afterUnmarshal*: callbacks called (when defined) before and after unmarshalling

6 Java RESTful service

6.1 RESTful web service architecture



6.2 @path annotation and resource URI I

- Each resource is identified by a URI
  - Server

```
http://localhost:9999/MyServer/
```

  - Class with a @path annotation for a RestFul java class

```
@Path("/hello") // http://localhost:9999/MyServer/hello
public class Hello { ...}
```

  - Method with a @path annotation

```
    @Path("replace") //http://localhost:9999/MyServer/hello/replace
    public String replace(...) {
    }
```

6.3 RestFul class recap table I

- It may help to build a recap table for each RestFul java class

method	name of the method
subpath	subpath
CRUD	create, read, update or delete
http msg	PUT, GET, POST, DELETE
arameters	additional parameters in the URL
esentation	format (e.g., json, XML, html, text)

- Example for class *Hello*, subpath hello, with one method *replace* (encore incomplet à ce stade du cours)

method	*SubPath*	*CRUD*	*http msg*	*parameters*	*presentation*
replace	replace	update	TBD	TBD	TBD

6.4 Input or output representation format I

- Defined with @consumes for input (POST and PUT) and @produces for output (GET)
- Defined for a class and/or overloaded on a method
- Client requirement and server representation offers should match

```
1 @Path("/skiers")
2 @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON, MediaType.TEXT_XML })
3 @Consumes({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
4 public final class SkiersRest implements SkiersInterface {
```

6.4 Input or output representation format II

- Client requirement defined in the GET request

```
1 GET /hello HTTP/1.1
2 Host: localhost
3 Accept: text/html, text/plain
```

- Service offeree

```
1     @GET
2     @Produces("text/html")
3     public String readHTML() {
4         return "<html><body>"+msg + "</body></html>";
5     }
```

- Recap table

method	*SubPath*	*CRUD*	*http msg*	*parameters*	*presentation*
readHTML	/	read	GET	/	HTML

- Matching representation defined in the response header

```
1 HTTP/1.1 200 OK
2 Content-Type: text/html
3
4 <html><body>Hello</body></html>
```

6.5 JAXB representation or JSON I

- Produces an XML or a JSON string (according to the client request)
- Return type is a class, the marshalling is done **automatically** by the jaxb or jackson library

```
1 @GET
2 @Path("searchskier")
3 //http://localhost:9999/MyServer/skiers/searchskier?name=xxx
4 @Produces({ MediaType.APPLICATION_XML, MediaType.APPLICATION_JSON})
5 public Skier getSkier(String name){
6     ...
7     Skier foundSkier= lookup(name);
8     return foundSkier; // marshalled in XML with JAXB or in Json with Jackson
9 }
```

method	*SubPath*	*CRUD*	*http msg*	*parameters*	*presentation*
getSkier	searchskier?name=	read	GET	name	XML or Json

6.6 Query parameters (GET) I

- Parameters : URL query parameter
  - Requested URL

```
http://localhost:9999/MyServer/calc/add?a=3&b=4
http://localhost:9999/MyServer/calc/add?a=3
```

  - Method definition

```
@Path("/calc")
public class CalcRest {
    @GET
    @Path("/add")
    @Produces(MediaType.TEXT_PLAIN)
    public String addPlainText(@QueryParam("a") double a,
    @DefaultValue("0") @QueryParam("b") double b) { return (a + b) + "";
    }
```

method	*SubPath*	*CRUD*	*http msg*	*parameters*	*presentation*
addPlainText	add?a=&b=	read	GET	a,b	TEXT

6.7 Path parameters I

- Parameters included in the path
  - Requested URL

```
http://localhost:9999/MyServer/calc/add/3/4
```

  - Method definition

```
@Path("/calc")
public class CalcRest {
    @GET
    @Path("/add/{a}/{b}")
    @Produces(MediaType.TEXT_PLAIN)
    public String addPlainText(@PathParam("a") double a,
    @DefaultValue("0") @PathParam("b") double b) { return (a + b) + "";
    }
```

method	*SubPath*	*CRUD*	*http msg*	*parameters*	*presentation*
addPlainText	add/a/b=	read	GET	a,b	TEXT

6.8 Other params

- *@FormParam*, Form parameters (POST)
- *@HeaderParam*, parameter extracted from the header
- *@CookieParam*, parameter extracted from the cookie

6.9 Hello World in REST I

(ExamplesREST/JAXREST-01) [6]

```
1  import javax.ws.rs.*;
2
3  @Path("/hello") // This is the base path, which can be extended at the method level.
4  public class HelloRest {
5      private static String msg = "Hello world";
6
7      public static void setMsg(final String msg) { HelloRest.msg = msg;}
8
9      @GET
10     @Produces("text/plain")
11     public String read() { return msg + "\n";}
12
13     @GET
14     @Produces("text/html")
15     public String readHTML() { return "<html><body>" + msg + "</body></html>";}
16
17     @GET
18     @Produces("text/plain")
19     @Path("/{extra}") // http:..../hello/xxx
20     public String personalizedRead(final @PathParam("extra") String cus) { return HelloRest.msg + ": " + cus + "\n";}
21
22     @GET
23     @Produces("text/plain")
24     @Path("replaceby") // http:..../hello/replaceby?newmsg=xxx
25     public String replaceBy(final @DefaultValue("") @QueryParam("newmsg") String newMsg) {
26         System.out.println("replaceBy new_msg=" + newMsg);
27         HelloRest.msg = newMsg;
28         return HelloRest.msg + "\n";
29     }
30
```

6.9 Hello World in REST II

```
31      @PUT
32      @Consumes("text/plain")
33      @Path("replace")
34      public void replace(final String newMsg) {
35          System.out.println("replace new_msg=" + newMsg);
36          HelloRest.msg = newMsg;
37      }
38
39      @DELETE
40      @Path("/delete")
41      public void delete() {
42          HelloRest.msg = "";
43          System.out.println("Message deleted.\n");
44      }
45  }
```

method	*SubPath*	*CRUD*	*http msg*	*parameters*	*presentation*
read	/	read	GET	/	TEXT
readHTML	/	read	GET	/	HTML
personalized_read	{extra}	read	GET	extra	TEXT
replaceBy	replaceby?newmsg=	read :-(	GET	newmsg	TEXT
replace	replace	update	PUT	newmsg	TEXT
delete	delete	delete	DELETE	/	/

6. *REST examples: directory REST-JAXREST-01*

6.10 Java Client example

(ExamplesREST/JAXREST-01)

```
1   restURI = "http://" + properties.getProperty("rest.serveraddress") + "/MyServer";
2   Client client = ClientBuilder.newClient();
3   URI uri = UriBuilder.fromUri(restURI).build();
4   WebTarget service = client.target(uri);
5   service.path("hello").path("replace").request().put(Entity.text("coucou"));
6   String getResult= service.path("hello").request().accept(MediaType.TEXT_PLAIN).get(String.class);
7   System.out.println("--------------------------------------------");
```

- path("hello"): subpath (or path parameters)
- request(): create an http request for the path
- accept(MediaType.TEXTPLAIN): accepted representation format
- get(String.class): message http GET, the return body is converted into a string

6.11 Light Grizzly server I

(ExamplesREST/JAXREST-01)

```
1         */
2        public static HttpServer startServer() throws IOException {
3                // server address defined in a property file
4                Properties properties = new Properties();
5                FileInputStream input = new FileInputStream("src/main/resources/rest.properties");
6                properties.load(input);
7                baseURI = "http://" + properties.getProperty("rest.serveraddress") + "/MyServer/";
8
9                // create a resource config that scans for JAX-RS resources and providers
10               // in the server package
11               final ResourceConfig rc = new ResourceConfig().packages("server");
12               // create and start a new instance of grizzly http server
13               // exposing the Jersey application at BASE_URI
14               return GrizzlyHttpServerFactory.createHttpServer(URI.create(baseURI), rc);
15        */
16       public static void main(final String[] args) throws IOException {
17               final HttpServer server = startServer();
18               System.out.println(String.format(
19                               "Jersey app started with WADL available at " + "%sapplication.wadl\nHit enter to stop it...", baseURI));
20               System.in.read();
21               server.shutdownNow();
22       }
```

- The server will handle requests for all the RestFul classes in the server package

7 REST Synthesis

-  Easy to write and easy to test RESTful WebServices and REST clients
  - As a consequence, a high percentage of deployed web services are RESTful services

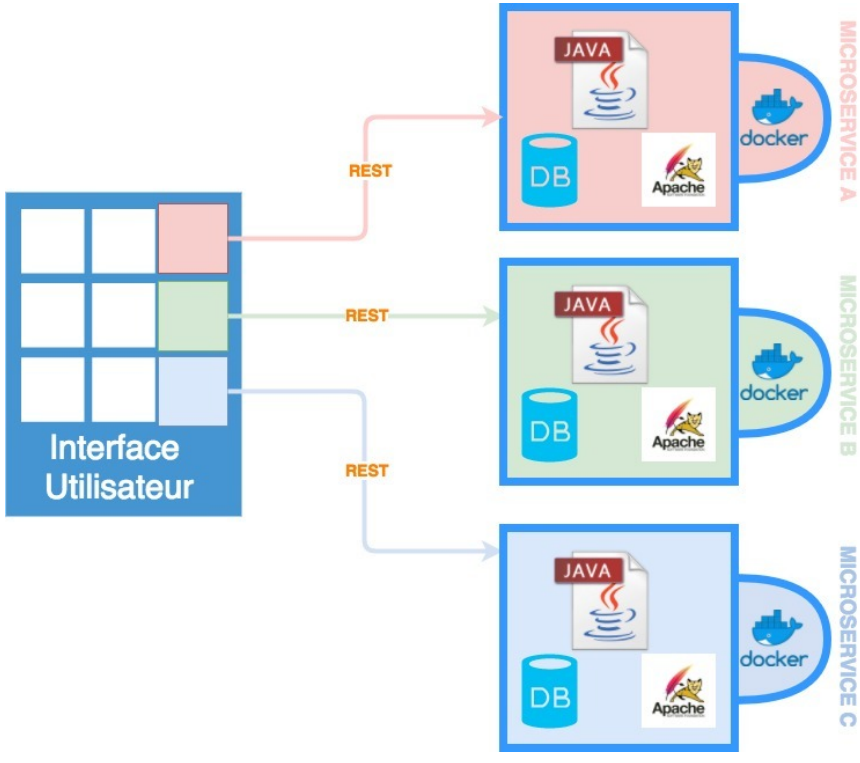
7 REST and microservices I

Microservice architecture

A microservice is a software architectural style that structures an application as a collection of loosely coupled services. Advantages:

Advantages

- modularity
- continuous delivery
- better scalability



7 REST and microservices II

Microservices interaction patterns

- Services in a microservice architecture are often processes that communicate over a network
  - For synchronous interactions: REST over HTTP (one of the most popular)
  - For Asynchronous interactions: AMQP and Akka actors are good candidates

7.1 Some links to be studied

- Json 2 java `http://www.jsonschema2pojo.org/`
- swagger and open API
  `https://swagger.io/docs/specification/about/`
  - A language to describe API
  - Tools to generate the skeleton of classes from an API description
  - Tools to generate the documentation of an API (example of generated documentation `https://www.versasense.com/docs/rest/`

References

Burke, B. (2010).

RESTful Java.

O'Reilly.

Fielding, R. T. (2000).

REST Architectural Styles and the Design of Network-based Software Architectures.

Doctoral dissertation, University of California, Irvine.

Kalin, M. (2010).

Java Web Services, Up and Running.

O'Reilly.