

JAX-RS = Api Java normalisée pour services REST

Les principales implémentations de JAX-RS sont :

- **Jersey**, implémentation de référence de SUN (bien : simple/léger et efficace + coté client)
- **Resteasy**, l'implémentation interne à jboss (bien)
- **CXF** (gérant à la fois "SOAP" et "REST" , remplacer si besoin la sous couche "jettison" par "jackson" pour mieux générer du "json")
- **Restlet** (???)

Principales annotations de JAX-RS

Pour implémenter les services REST, on utilise principalement les annotations JAX-RS suivantes:

- **@Path** : définit le chemin (fin d'URL) de la ressource.
Cette annotation se place sur la classe du service et/ou sur une de ses méthodes
- **@GET, @PUT, @POST, @DELETE** : définit le mode HTTP (selon logique CRUD des méthodes)
- **@Produces** spécifie le ou les Types MIME de la réponse du service
- **@Consumes** : spécifie le ou les Types MIME acceptés en entrée du service

NB: en **version 1** de JAX-RS, ces annotations sont à placer sur la **classe d'implémentation** .

Depuis la version 2 de JAX-RS, on peut également **les placer** sur une **interface java** (*avec éventuelle répétition si besoin coté classe*).

Classe de données compatible JAX-RS (en mode Xml)

```
package pojo;  
import javax.xml.bind.annotation.XmlRootElement;
```

```
/* pour préciser balise xml englobante et pour jaxb2  
(ceci n'est utile que pour un transfert XML et n'est  
généralement pas nécessaire pour un transfert JSON) */
```

```
@XmlRootElement(name = "user")
```


```
public class User {  
    private Integer id;    // +get/set  
    private String name;  // +get/set
```

```
    @Override  
    public String toString() {  
        return String.format("{id=%s,name=%s}", id, name);  
    }  
}
```

Classe d'implémentation d'un Web-service JAX-RS

```
package service;
import java.util.HashMap;          import java.util.Map;
import javax.ws.rs.GET;           import javax.ws.rs.Path;
import javax.ws.rs.PathParam;    import javax.ws.rs.QueryParam;
import javax.ws.rs.Produces;     import javax.ws.rs.core.Response;
import javax.ws.rs.core.Response.Status;

@Path("myservice") //partie de l'url (préfixe commun à toutes méthodes)
@Produces("application/json") //par défaut pour toutes les méthodes
                                //de la classe
//+ éventuel @Named selon contexte (Spring ou CDI/JEE6 ou ...)
public class ServiceImpl /* implements ServiceRestItf */{
    ...
}
```



*Pas d'interface à obligatoirement implémenter avec JAX-RS v1.
Implémenter une interface java (comportant des annotations de JAX-RS)
est cependant possible et conseillé depuis JAX-RS 2 et JEE7 .*

implémentation JAX-RS de recherche par id

```
public class ServiceImpl /* implements ServiceRestItf */{

    private static Map<Integer,User> users = new HashMap<Integer,User>();

    static { //jeux de données (pour simulation de données en base)
        users.put(1, new User(1, "foo"));    users.put(2, new User(2, "bar"));
    }

    // + éventuel injection d'un service local interne pour déléguer les appels :
    // @Autowired ou @Inject ou @EJB ou ....
    // private ServiceInterne serviceInterne ;

    @GET
    @Path("users/{id}")
    // pour URL = http://localhost:8080/mywebapp/services/rest/myservice/users/2
    public User getUser(@PathParam("id") Integer id) {
        return users.get(id);
    }

    .../...
```

Implémentation JAX-RS de recherches multiples

```
public class ServiceImpl /* implements ServiceRestItf */{ ...
```

```
    private Collection<User> getAllUsers() {  
        return users.values();  
    }
```

```
    @GET
```

```
    @Path("users")
```

```
    // http://localhost:8080/mywebapp/services/rest/myservice/users?name=foo
```

```
    // et quelquefois ...?p1=val1&p2=val2&p3=val3
```

```
    public User getUserByCriteria(@QueryParam("name") String name) {  
        if(name!=null) { ...; }  
        else { return getAllUsers() ; }  
    }
```

```
    @GET
```

```
    @Path("bad")
```

```
    public Response getBadRequest() {
```

```
        return Response.status(Status.BAD_REQUEST).build();
```

```
        //le comble d'un service est de ne rendre aucun service !!!! (bad: pas bien: is no good) !!!
```

```
    }
```

implémentation JAX-RS d'envoi de données (v1)

```
public class ServiceImpl /* implements ServiceRestItf */ {    ...
    @POST    // (REST recommande fortement POST pour des ajouts )
    @Path("users")
    // pour URL = http://localhost:8080/mywebapp/services/rest/myservice/users
    // dans form avec <input name="id" /> et <input name="name" /> et method="POST"
    public Response addNewUser(@FormParam("id") Integer id,
                               @FormParam("name") String name) {
        users.put(id,new User(id,name));
        return Response.status(Status.OK).build();
    }

    @PUT
    @Path("users")
    public Response updateUser(@FormParam("name")String newName,
                               @FormParam("id")Integer id){
        User user = users.get(id);
        if(user!=null){
            user.setName(newName);
            return Response.status(Status.OK).build();
        }
        else return Response.status(Status.NOT_FOUND).build();
    }
}
```

Retour en text/plain pour opérations élémentaires

```
@GET
@Path("/euroToFranc/{s}")
@Produces("text/plain")
public double euroToFranc(@PathParam("s")double s){
    return s*6.55957 ;
}
```

http://localhost:8080/xx/yy/euroToFranc/15 ---> 98.39355

@Produces("application/xml") ou **@Produces("application/json")**

Combiné avec code java unique

@GET

@Path("users/{id}")

// pour URL = http://localhost:8080/mywebapp/services/rest/myservice/users/2

public User getUser(@PathParam("id") Integer id) {

 return users.get(id);

}

Pour que le framework JAX-RS
puisse automatiquement générer et retourner

```
<?xml version="1.0" encoding="UTF-8"
      standalone="yes"?>
```

```
<user>
```

```
  <id>1</id>
```

```
  <name>foo</name>
```

```
</user>
```

Conversion automatique java → xml

```
{
  "id" : 1,
  "name" : "foo"
}
```

*Conversion automatique
java → json*

@Produces et @Consumes en "application/json" (pour Angular-js , ...)

@Path("/json/")

@Produces("application/json")

@Consumes("application/json")

public class **ProductRestJsonService** {

...

private ProductService **prodService** = new ProductService(); *//may be injected by spring*

@POST *//POST --> save/create or saveOrUpdate dans logique "Angular-Js"*

@Path("products/{id}")

//p est passé comme un seul bloc (en mode json) dans le corps/body de la requête

//exemple: { id: 1 , name='xxx' , label='yyy' , ...}

public **Product** **saveOrUpdateProduct**(**@PathParam("id")**Long id,**Product** p){

 if(id!= null){

 prodService.updateProduct(p);

 }

 else{

 Long newId= prodService.addNewProduct(p);


 p.setId(newId); *//auto_incr id/pk*

 }

 return p;

 }

}



@Consumes permet une conversion automatique **json → java** en entrée .

Configuration de JAX-RS intégrée à un projet JEE6/CDI

Lorsque JAX-RS est utilisé au sein d'un projet JEE6/CDI (par exemple avec ou sans "EJB3" pour JBoss7 ou bien pour Tomcat EE) , on peut configurer la partie intermédiaire des URL "rest" et les classes java à prendre en charge via une configuration ressemblant à la suivante :

```
package tp.web.rest; //ou autre
import java.util.HashSet; import java.util.Set;
import javax.ws.rs.ApplicationPath; import javax.ws.rs.core.Application;
```

```
//pour url en http://localhost:8080/myWebApp/services/rest + @Path() java
@ApplicationPath("/services/rest")
public class MyRestApplicationConfig extends Application {
```

```
    @Override
```

```
    public Set<Class<?>> getClasses() {
```

```
        final Set<Class<?>> classes = new HashSet<Class<?>>();
```

```
        // register root resource(s):
```

```
        classes.add(XyServiceRest.class);
```

```
        classes.add(ServiceRest2.class);
```

```
        return classes;
```

```
    }
```

```
} //Ceci fonctionne avec des classes java (ex: XyServiceRest) annotées par "@Named"
```

```
// et avec beans.xml présent dans WEB-INF .
```

Configuration de JAX-RS avec CXF intégré dans Spring (1/2)

Beaucoup de versions de CXF utilisent par défaut "jettison" en interne pour générer du "json", ce qui pose des problèmes de compatibilité avec Angular-Js (ou ...).

Il est conseillé de configurer CXF pour qu'il utilise "jackson" à la place de "jettison".

CXF 2.x est compatible avec jackson 1.9 (*org.codehaus.jackson*)

CXF 3.x est plutôt compatible avec jackson 2.x (*com.fasterxml.jackson*)

WEB-INF/web.xml (*spring+cxf*) :

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/classes/beans.xml</param-value>
    <!-- ou .... , chemin menant à la configuration spring -->
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener
    </listener-class> <!-- initialisation de spring lors dès le démarrage webApp -->
</listener>

<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
</servlet>    <!-- cxf...jar recherchés dans WEB-INF/lib ou ... -->
<servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern> <!-- ou autre que services/* →
    <!-- URL WS en "http://localhost:8080/myWebApp/services/..." -->
</servlet-mapping>
```

Configuration de JAX-RS avec CXF intégré dans Spring (2/2)

beans.xml (*spring+cxf*)

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:jaxws="http://cxf.apache.org/jaxws" xmlns:jaxrs="http://cxf.apache.org/jaxrs"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd
http://cxf.apache.org/jaxrs http://cxf.apache.org/schemas/jaxrs.xsd">
<import resource="classpath:META-INF/cxf/cxf.xml" />
<bean id='jacksonJsonProvider' class='com.fasterxml.jackson.jaxrs.json.JacksonJaxbJsonProvider' />
<bean id='jacksonXmlProvider' class='com.fasterxml.jackson.jaxrs.xml.JacksonJaxbXMLProvider' />

<!-- url complete de type "http://localhost:8080/mywebapp/services/rest/myservice/users/"
avec "services" associe à l'url-pattern de CxfServlet dans web.xml
et myservice/users associé aux valeurs de @Path() de la classe java et des méthodes -->
<jaxrs:server id="myRestServices" address="/rest">
    <jaxrs:providers>
        <ref bean='jacksonJsonProvider' /> <ref bean='jacksonXmlProvider' />
    </jaxrs:providers>
    <jaxrs:serviceBeans>
        <ref bean="serviceRestImpl" />        <!-- <ref bean="service2Impl" /> -->
    </jaxrs:serviceBeans>
</jaxrs:server>

<bean id="serviceRestImpl" class="service.ServiceRestImpl" /> </beans>
```

Configuration maven type pour cxf (dans pom.xml)

```
<properties>
```

```
  <org.springframework.version>4.1.1.RELEASE</org.springframework.version>
```

```
  <org.apache.cxf.version>3.0.2</org.apache.cxf.version>
```

```
</properties>
```

```
<dependencies>
```

```
  <dependency>
```

```
    <groupId>org.apache.cxf</groupId>
```

```
    <artifactId>cxf-api</artifactId>
```

```
    <version>${org.apache.cxf.version}</version>
```

```
  </dependency>
```

```
  <!-- idem pour les artifactId suivants :
```

```
  cxf-rt-frontend-jaxrs , cxf-rt-frontend-jaxws
```

```
  cxf-rt-ws-security ,    cxf-rt-transport-http    -->
```

```
  <dependency>
```

```
    <groupId>com.fasterxml.jackson.jaxrs</groupId>
```

```
    <artifactId>jackson-jaxrs-json-provider</artifactId>
```

```
  <!-- <artifactId>jackson-jaxrs-xml-provider</artifactId> -->
```

```
    <version>2.2.3</version>
```

```
  </dependency>
```

```
</dependencies>
```

Configuration de JAX-RS avec jersey

Dans pom.xml :

```
<dependency>
  <groupId>org.glassfish.jersey.containers</groupId>
  <artifactId>jersey-container-servlet-core</artifactId> <version>2.11</version>
</dependency>
<dependency>
  <groupId>org.glassfish.jersey.media</groupId>
  <artifactId>jersey-media-json-jackson</artifactId> <version>2.11</version>
</dependency>
```

Dans WEB-INF/web.xml (jersey) :

```
<servlet>
  <servlet-name>Jersey_REST_Service</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <!-- package java avec classes à prendre en charge -->
    <param-value>tp.service.rest</param-value>
  </init-param>
</servlet>
<servlet-mapping>
  <servlet-name>Jersey_REST_Service</servlet-name>
  <url-pattern>/services/rest/*</url-pattern>
</servlet-mapping>
```

Appel de webServices REST en java (vue d'ensemble des possibilités)

La **version 1** de la spécification **JAX-RS** a normalisé des annotations pour implémenter un WebService REST coté serveur mais n'avait **rien prévu pour le coté client**.

Pour le coté client des services web REST en java on peut utiliser une **API** open-source du monde *apache* spécialisée dans les appels http (en modes GET, POST, PUT, DELETE) : **httpclient** (de httpcomponents).

La technologie "**jersey**" a dès les premières versions proposé (hors spécifications JAX-RS 1) une **api cliente** pour appeler des WS REST.

Plus récemment, la version 2 des spécifications JAX-RS a normalisé :

- une **api java cliente et standardisée** (pour invoquer WS REST)
- la notion d'**interface java de WebService REST** (avec annotations JAX-RS) **que l'on peut réutiliser (par copie) du coté client** de façon à **générer dynamiquement un proxy "java/objet/rpc"** vers le webService REST distant.

Appel de webServices REST en java via l'API "**httpclient**" (1)

Pour le coté client des services web REST en java on peut utiliser une **API** open-source du monde *apache* spécialisée dans les appels http (en modes GET, POST,PUT, DELETE) : **httpclient** (de httpcomponents).

```
<dependency>
  <groupId>org.apache.httpcomponents</groupId>
  <artifactId>httpclient</artifactId><!-- indirectly httpcore -->
  <version>4.2.1</version>
</dependency>
```

```
package client;

import java.net.URI;
import org.apache.http.HttpResponse;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.client.utils.URIBuilder;
import org.apache.http.impl.client.DefaultHttpClient;
import org.apache.http.util.EntityUtils;

public class CalculeurClientRestApp {

    public static void main(String[] args) {
        .../...
```

Appel de webServices REST en java via l'API "**httpClient**" (2)

```
try {
    String restAppPart = "/wsCalculateur/services/rest";
    URIBuilder builder = new URIBuilder();
    builder.setScheme("http").setHost("localhost").setPort(8080)
        .setPath(restAppPart + "/calculateur/addition")
        .setParameter("a", "5")
        .setParameter("b", "6");
    URI uri = builder.build();
    String url5plus6=uri.toString();
    System.out.println("REST GET URL="+url5plus6);

    HttpClient httpClient = new DefaultHttpClient();
    HttpGet httpget = new HttpGet(url5plus6);
    HttpResponse response = httpClient.execute(httpget);

    String res5plus6=EntityUtils.toString(response.getEntity());

    System.out.println("5+6="+res5plus6);
} catch (Exception e) {
    e.printStackTrace();
}
```

Code brut (à encapsuler dans des classes utilitaires) !

Appel de webServices REST en java via l'API "[httpclient](#)" (3)

```
public static void testCreateNewDevise(String name,String change) {
    try {
        ...
        HttpPost httppost = new HttpPost(urlCreateDevises);

        List<NameValuePair> formparams = new ArrayList<NameValuePair>();
        formparams.add(new BasicNameValuePair("name", name));
        formparams.add(new BasicNameValuePair("change", change));
        UrlEncodedFormEntity entity=new UrlEncodedFormEntity(formparams,"UTF-8");
        httppost.setEntity(entity);
        HttpResponse response = httpclient.execute(httppost);
        System.out.println("response:"+response.toString());
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public static void testUpdateDevise(String name,String change) {
    ...
    HttpPut httpput = new HttpPut(urlUpdateDevises);
    ...
    UrlEncodedFormEntity entity =new UrlEncodedFormEntity(formparams,"UTF-8");
    httpput.setEntity(entity);
    HttpResponse response = httpclient.execute(httpput);
    System.out.println("response:"+response.toString());
    ...
}
```

Appel de webServices REST en java via l'API "**httpclient**" (4)

```
public static void testDeleteDevise(String name) {
    String restAppPart = "/wsCalculateur/services/rest";
    UriBuilder builder = new UriBuilder();
    builder.setScheme("http").setHost("localhost").setPort(8080)
        .setPath(restAppPart + "/devises/delete/" + name);
    URI uri = builder.build();
    String urlDeleteDevises=uri.toString();
    HttpClient httpClient = new DefaultHttpClient();
    HttpDelete httpdelete = new HttpDelete(urlDeleteDevises);
    HttpResponse response = httpClient.execute(httpdelete);

    System.out.println("response:"+response.toString());
    ...
}
```

Autre possibilité dès JAX-RS 1 : L'implémentation "**Jersey**" de l'api "jax-rs" offre quelques classes permettant d'invoquer un service REST (coté "**client**").

Appel de webServices REST en java via l'API cliente de JAX-RS 2

Exemple simple (utilisant l'api standardisée) :

```
import javax.ws.rs.client.Client; import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity; import javax.ws.rs.client.Invocation;
import javax.ws.rs.client.WebTarget;
...
Client jaxrs2client = ClientBuilder.newClient();

String calculateurRestUrl =
    "http://localhost:8080/wsCalculateur/services/rest/calculateur";
double a=5.0 , b=6.0;
System.out.println("appel de " + calculateurRestUrl + "/addition?a=5&b=6");
WebTarget additionTarget = jaxrs2client.target(calculateurRestUrl)
    .path("addition")
    .queryParam("a", a)
    .queryParam("b", b);

double resAdd = Double.parseDouble(
additionTarget.request(MediaType.TEXT_PLAIN_TYPE)
    .get().readEntity(String.class));
System.out.println("\t 5+6=" + resAdd);
```

Appel de webServices REST en java via l'API cliente de JAX-RS 2 (suite)

Exemple en mode **POST** (au format **JSON**) :

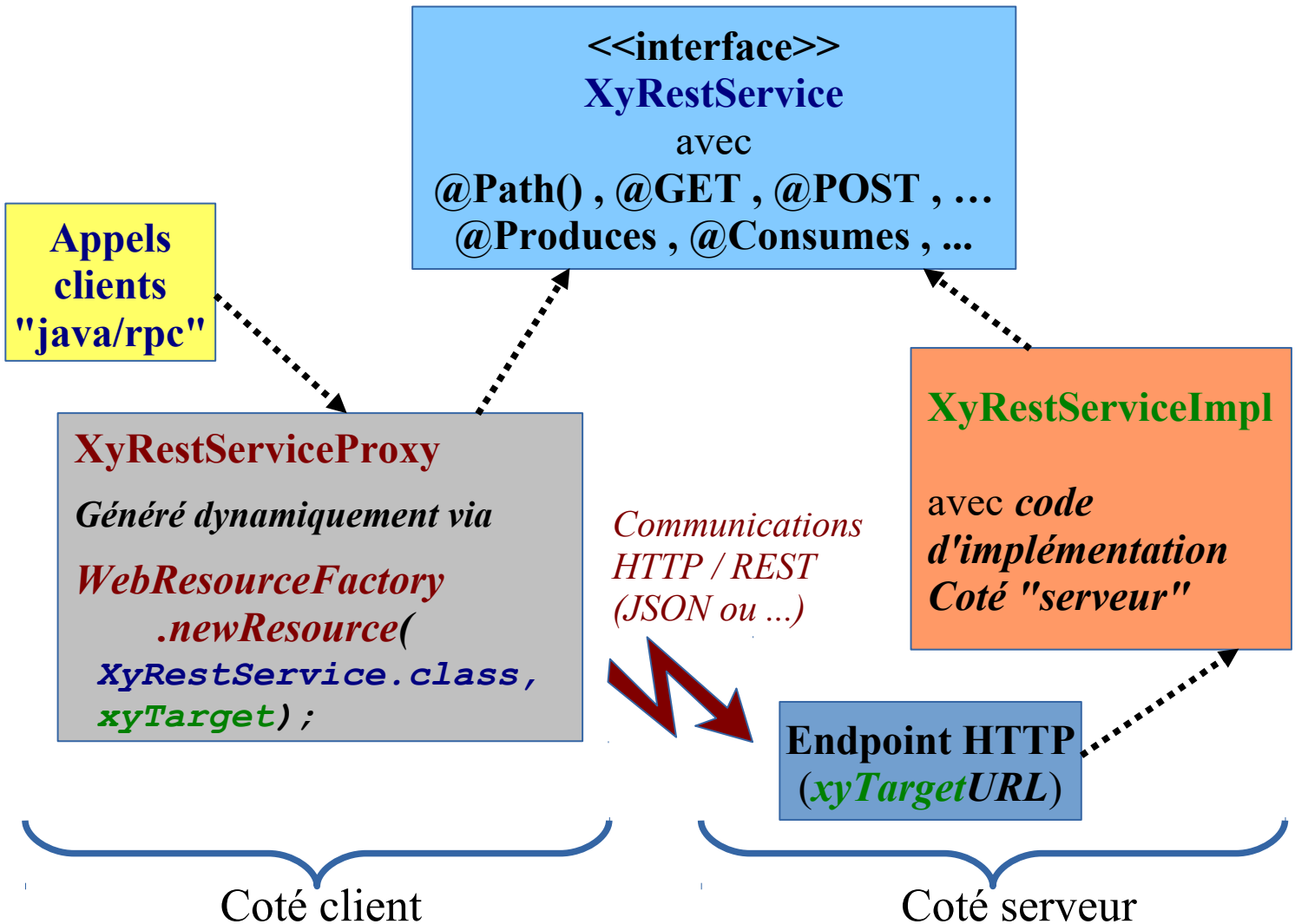
```
Long pk=null;          Product savedProd =null;
Product newProd = new tp.data.Product(null,"prodXy",
                                         "prod Xy with good features",12.89f);

Client jaxrs2client = ClientBuilder.newClient().register(JacksonFeature.class);
WebTarget productsTarget = jaxrs2client.target(
    "http://localhost:8080/wsCalculateur/services/rest/json/products");

Response responseSaveNewProduct = productsTarget
    .path("/0") //0 or null as pk for "POST" as "save new /insert" (not "update")
    .request(MediaType.APPLICATION_JSON_TYPE)
    .post(Entity.entity(newProd, MediaType.APPLICATION_JSON_TYPE));

if(responseSaveNewProduct.getStatus()==200 /*OK*/) {
    //String savedProductAsJsonString = responseSaveNewProduct.readEntity(String.class);
    savedProd = responseSaveNewProduct.readEntity(Product.class);
    pk=savedProd.getId();
    System.out.println("(saved) new product with auto_incremented pk = " + pk
        + "\n\t " + savedProd.toString());
} else {
    System.err.println(responseSaveNewProduct);
}
```

Appel de webServices REST avec **proxy JAX-RS 2** (principe)



Appel de webServices REST avec **proxy JAX-RS 2** (exemple)

```
import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.WebTarget;
import org.glassfish.jersey.client.proxy.WebResourceFactory;
import org.glassfish.jersey.jackson.JacksonFeature;
import tp.service.ProductRestJsonService; //interface avec @Annot JAX-RS2
...
Client jaxrs2client = ClientBuilder.newClient()
                                .register(JacksonFeature.class);
WebTarget productsTarget = jaxrs2client.target(
    "http://localhost:8080/wsCalculateur/services/rest/json/products");

// create a new client proxy for the ProductRestJsonService :
ProductRestJsonService productRestJsonProxyService =
    WebResourceFactory.newResource(ProductRestJsonService.class,
                                    productsTarget);

Product prod1 = productRestJsonProxyService.getProductById(1L);
System.out.println("prod1:" + prod1.toString());

//Test creation/insert:
Product newProd = new tp.data.Product(null,"prodXy",
                                       "prod Xy with good features",12.89f);
savedProd = productRestJsonProxyService
            .saveOrUpdateProduct(0L, newProd); //0L for new to saved
System.out.println("new product saved : " + savedProd);
```


Appel de webServices REST depuis javascript avec jQuery et ajax

```
jQuery.ajax({  
  type: "GET | POST | DELETE | PUT",  
  url: url,  
  data: data,  
  dataType: "text | html | json | jsonp | script | xml"  
  success: success_callback,  
  error: error_callback  
});
```

```
function updateProduct (product) {
```

```
  jQuery.ajax({  
    type: "PUT",  
    url: "http://localhost:8080/xx.yy/products",  
    contentType: "application/json; charset=utf-8",  
    data: JSON.stringify(product),  
    dataType: "json",  
    success: function (data, status, jqXHR) { // do something  
    },  
    error: function (jqXHR, status) { // error handler  
    }  
  });  
}
```

exemple

Alternative "Spring-MVC" (vis à vis de JAX-RS)

Dans le monde **java** , il est possible de programmer un web service "REST" avec **Spring-MVC** à la place de JAX-RS.

Inconvénients de Spring-MVC (pour WS REST):

- * ***ce n'est pas le standard officiel*** , c'est du ***spécifique "Spring"***
... sachant que le standard s'est maintenant amélioré en version 2 ...
- * la technologie concurrente JAX-RS s'intègre pas trop mal dans Spring (via jersey ou cxf).

Avantages de Spring-MVC (pour WS REST) :

- * **l'intégration au sein d'un projet "Spring" est plus aisée (configuration plus simple , moins de librairies ".jar" nécessaires , ...)**
- * **facilement combinable avec d'autres extensions de Spring (spring-security , spring-boot , ...)**
- * logique "MVC" pouvant être utile si l'on retourne des portions d'HTML.

Au final, si projet JEE/CDI → standard JAX-RS
si projet Spring → Spring-MVC ou bien JAX-RS

Exemple de WS-REST codé avec "Spring-MVC"

@RestController

@RequestMapping(value="/devises" , headers="Accept=application/json")

public class *DeviseListCtrl* {

@Inject

private GestionDevises serviceDevises; *//internal Spring local service*

@RequestMapping(value="/" , method=RequestMethod.GET)

@ResponseBody

List<Devise> *getAllDevises()* {

return serviceDevises.getListeDevises();

}

@RequestMapping(value="/{name}" , method=RequestMethod.GET)

@ResponseBody

Devise *getDeviseByName(@PathVariable("name") String deviseName)* {

return serviceDevises.getDeviseByName(deviseName);

}

}

- même logique de paramétrage que JAX-RS (HTTP method , path , ...)
- annotations différentes de celles de JAX-RS (avec davantage d'attributs)
- quasiment aucune configuration annexe nécessaire (avec java-config , spring-boot)

Notion d' API REST

Dès le début , la structure d'un web-service "*SOAP*" a été décrite de façon standardisée via la norme *WSDL* (standard officiel du W3C).

A l'inverse les Web-services REST (qui sont basés sur de simples recommandations autour de l'usage d'HTTP) ne sont toujours pas associés à un type de description unique et standardisé.

Un **document qui décrit la structure d'un web-service REST** est généralement appelé "**API REST**" et peut être écrit en XML , en JSON ou en YAML.

Les principaux formalismes existants pour décrire une API REST sont :

- **WADL** (existant depuis longtemps en XML mais en perte de vitesse)
- **Swagger** (version 1.x basée sur YAML , v2 basée sur JSON que l'on peut convertir en YAML). Bien outillé et existant depuis plusieurs années, swagger a pour l'instant une petite longueur d'avance.
- **RAML** (pour l'instant basé sur une syntaxe YAML volontairement simple , pris en charge par quelques marques telles que "MuleSoft" , ...)
- **Blueprint API** : basé également sur YAML

Format YAML

YAML (*YAML Ain't Markup Language*) n'est d'après son nom , pas un langage à balise mais se veut être un équivalent d'un point de vue fonctionnalité (à la fois sérialisation/dé-sérialisation de documents informatiques arborescents).

books.yaml (exemple)

YAML est en fait
structuré via des indentations et se
veut être **facile à lire**
(ou à écrire) par une
personne humaine .

*De la rigueur
dans les indentations
S'impose !!!*

(fragile comme CSV)

```
/books:
  /{bookTitle}
  get:
    queryParameters:
      author:
        displayName: Author
        type: string
        description: An author's full name
        example: Mary Roach
        required: false
      publicationYear:
        displayName: Pub Year
        type: number
        description: The year released for the first time
    ...
```

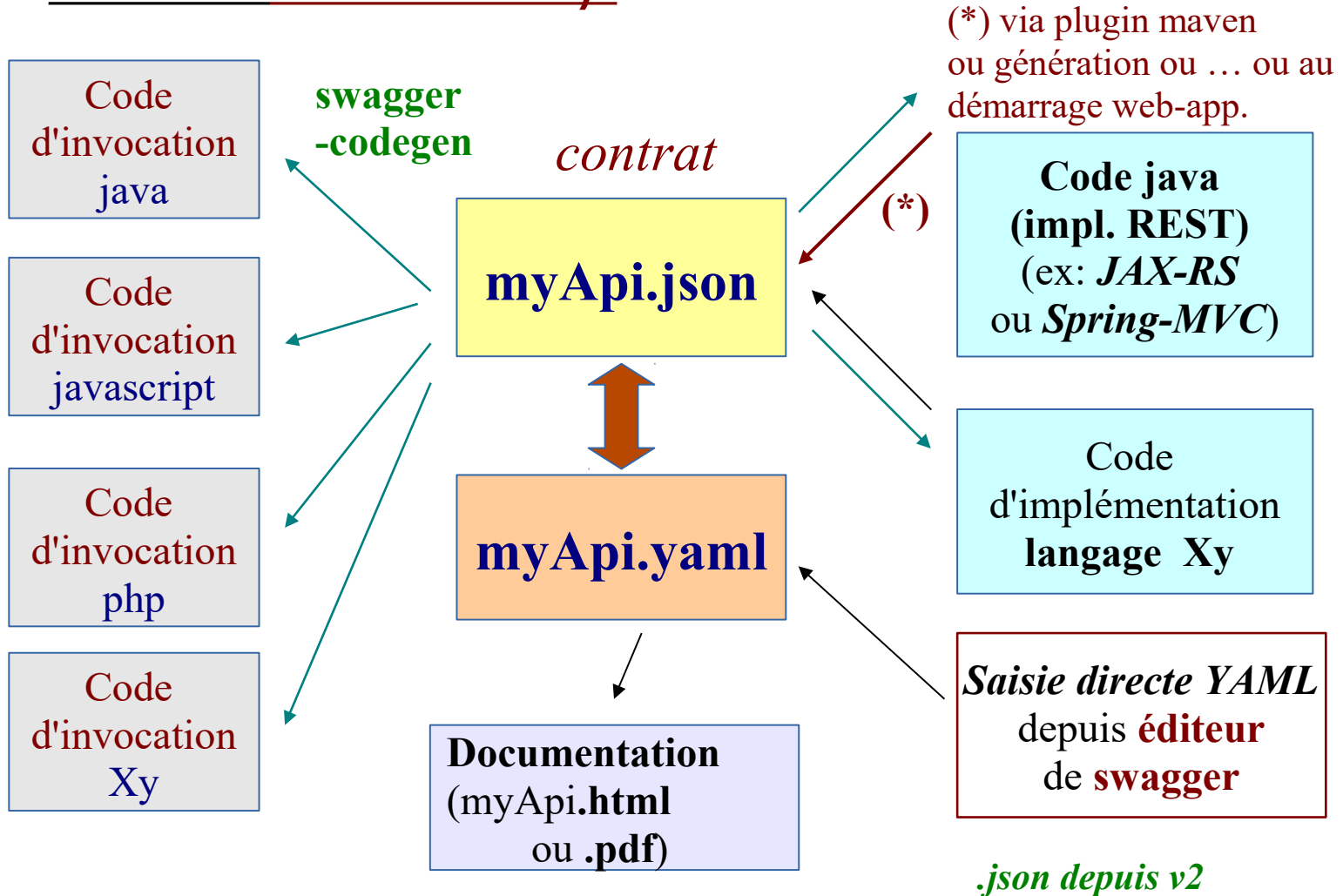
Que choisir entre Swagger , RAML , Blueprint-Api , ... ?

- ??? (l'avenir le dira)
- à court terme : tester un peu tout et garder ce qui semble le plus pratique .

Faut-il attendre un standard officiel et unique ?

- **Non** (car on peut attendre encore longtemps un standard qui ne viendra peut être pas ou qui viendra tardivement) .
- **Des passerelles sont déjà (et seront encore plus) disponibles pour passer d'un format à un autre .**

Swagger (spécifications pour **API REST** accompagnées d'une riche "boîte à outils").



```

<plugin>
  <groupId>com.github.kongchen</groupId>
  <artifactId>swagger-maven-plugin</artifactId>
  <version>1.5.3</version>
  <configuration>
    <apiSources>
      <apiSource> <!-- springmvc = false means JAX-RS -->
        <springmvc>false</springmvc>
        <locations>tp.service.RestDeviseService</locations>
        <!-- package(s) or classe(s) separator=; -->
        <schemes>http,https</schemes>    <host>localhost:8080</host>
        <basePath>/wsCalculateur/services/rest</basePath>
      </apiSource>
      <!-- <securityDefinitions>
        <securityDefinition> <name>basicAuth</name>
          <type>basic</type> </securityDefinition>
        <securityDefinition>
          <json>/securityDefinition.json</json>
        </securityDefinition>
      </securityDefinitions> -->
      <templatePath>${basedir}/src/test/resources/swagger-template/strapdown.html.hbs</templatePath>
      <outputPath>${basedir}/swagger-generated/document.html</outputPath>
      <swaggerDirectory>${basedir}/swagger-generated/swagger-ui</swaggerDirectory>
    </apiSources>
  </configuration>
</plugin>

```

```

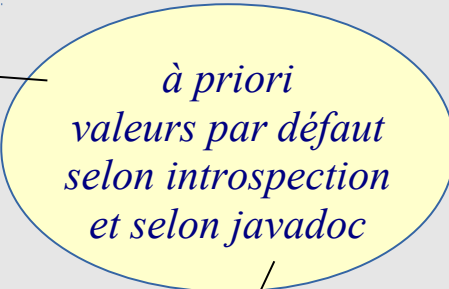
<dependency>
  <groupId>io.swagger</groupId>
  <artifactId>swagger-core</artifactId>
  <scope>compile</scope>
  <version>1.5.3</version>
  <exclusions>
    <exclusion>
      <groupId>javax.ws.rs</groupId>
    </exclusion>
  </exclusions>
</dependency>

```

**config
maven
swagger**

Annotations supplémentaires "swagger" (vis à vis de JAX-RS)

```
import io.swagger.annotations.Api;  
import io.swagger.annotations.ApiOperation;  
/**  
 * Classe d'implémentation du web-service "devises"  
 **/  
@Api(value = "/devises/")  
@Path("/devises/")  
public class RestDeviseService {  
    /**  
     * recherche par nom de devise  
     **/  
    @GET  
    @Path("/{name}")  
    @Produces("application/xml")  
    @ApiOperation(value = "Find Devise by name",  
        notes = "Returns a Devise", response = Devise.class)  
    public Devise getDeviseByName(@PathParam("name")String name){  
        return mapDevises.get(name);  
    }  
    ...  
}
```

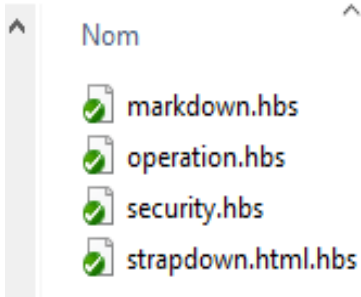


à priori
valeurs par défaut
selon introspection
et selon javadoc

The diagram consists of a yellow oval with a blue border. Inside the oval, the text "à priori", "valeurs par défaut", "selon introspection", and "et selon javadoc" is written in a blue, italicized font. Two arrows originate from this oval: one points to the **@Api** annotation in the code, and the other points to the **@ApiOperation** annotation.

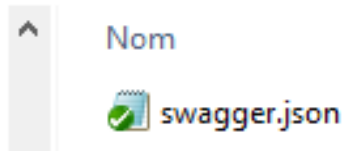
Templates à recopier/préparer :

resources > swagger-template

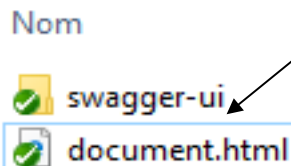


mvn swagger-maven-plugin:generate

swagger-generated > swagger-ui



> swagger-generated



*Documentation
html de l'api REST
(agréable à lire)*

```
{
  "swagger": "2.0",
  "host": "localhost:8080",
  "basePath": "/wsCalculateur/services/rest",
  "tags": [ { "name": "calculateur" }, { "name": "devises" } ],
  "schemes": [ "http", "https" ],
  "paths": {
    "/devises/{name}" : {
      "get" : {
        "tags": [ "devises" ],
        "summary": "Find Devise by name",
        "description": "Returns a Devise",
        "operationId": "getDeviseByName",
        "produces": [ "application/xml" ],
        "parameters": [ { "name": "name", "in": "path",
          "required": true, "type": "string" } ],
        "responses": {
          "200": {
            "description": "successful operation",
            "schema": {
              "$ref": "#/definitions/Devise"
            }
          }
        }
      }
    }
  },
  "definitions": {
    "Devise": {
      "type": "object",
      "properties": {
        "name": { "type": "string" },
        "change": { "type": "number", "format": "double" }
      },
      "xml": { "name": "devise" }
    }
  }
}
```

→ à exploiter via **swagger-codegen**

Génération swagger automatique possible au démarrage de l'appli. web

Exemple avec implémentation Jersey de JAX-RS :

```
<servlet>
  <servlet-name>Jersey_REST_Service</servlet-name>
  <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
  <init-param>
    <param-name>jersey.config.server.provider.packages</param-name>
    <param-value>io.swagger.jaxrs.listing, tp.service.rest</param-value>
  </init-param>  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>Jersey_REST_Service</servlet-name>
  <url-pattern>/services/rest/*</url-pattern>
</servlet-mapping>

<servlet>
  <servlet-name>Jersey2Config</servlet-name>
  <servlet-class>io.swagger.jersey.config.JerseyJaxrsConfig</servlet-class>
  <init-param>  <param-name>api.version</param-name>
    <param-value>1.0.0</param-value>  </init-param>
  <init-param> <param-name>swagger.api.basepath</param-name>
    <param-value>http://localhost:8080/wsRestJersey/services/rest</param-value>
  </init-param>  <load-on-startup>2</load-on-startup>
</servlet>
```

*Descriptions dynamiquement
générées (au bout de l'url
relative **services/rest**) :*

swagger.json
swagger.yaml

Utilisation de "swagger-codegen"

Installation de "swagger-codegen" :

```
git clone https://github.com/swagger-api/swagger-codegen
```

```
cd swagger-codegen
```

```
./run-in-docker.sh mvn package
```

ou bien

```
mvn -skipTests package    # opération longue !
```

Utilisation "swagger-codegen" (en partant d'un exemple fourni) :

```
java -jar modules/swagger-codegen-cli/target/swagger-codegen-cli.jar generate \
-i http://petstore.swagger.io/v2/swagger.json \
-l java \
-o samples/client/petstore/java
```

-i (input) , -o (output) , -l (language)