

# JAX-RS czyli REST w Javie

Adam Kędziora

# Webservice

Usługa sieciowa (ang. web service) – komponent programowy niezależny od platformy i implementacji, dostarczający określonej funkcjonalności.

SOAP, UDDI, XML, WSDL - „tradycyjne” usługi sieciowe

# REST

REpresentational State Transfer – wzorzec architektury oprogramowania sieciowego w którym główną jednostką jest zasób.

Klasycznym przykładem jest protokół http w swojej oryginalnej postaci i oryginalnym przeznaczeniu – tak więc „rest” nie jest niczym nowym – to powrót do pierwotnych idei sieciowych.

# Co REST nam daje ?

## A) przenośność

Usługi sieciowe opracowywane w oparciu o wsdl/soap/uddi są ciężkie do implementacji jeżeli nie skorzysta się z dodatkowych narzędzi – przez co dużo trudniej zastosować je chociażby w przypadku rynku embeded. Usługi restowe – wymagają jedynie obsługi protokołu http – tak do tworzenia usług jak i ich konsumpcji.

# Co REST nam daje ?

## B) skalowalność

Duża zaleta usług rest wynika z przenoszenie stanu w wywołaniu http – jeżeli stan jest przenoszony przy wywołaniu, nie trzeba go specjalnie trzymać na serwerze – co oznacza dla programisty javy możliwość wykorzystania bezstanowych ziaren ejb – co oznacza wysoką skalowalność wszcz.

# REST w javie czyli JAX-RS

JAX-RS – jsr311  
Java API for RESTful Web Services

Aktualnie w wersji 1.1 – specyfikacja definiuje sposoby pisania serwisów – jednak nie ich konsumpcji, planowane jest to na wersję 2.0

```
@Path(/hello)  
Public class restHello{  
    @GET  
    Public String sayHello(){return „hello”;}   
}
```

# REST w javie czyli jersey

Jersey – referencyjna implementacja JAX-RS  
ś.p. Sun microsystems.

Referencyjna implementacja jax-rs bardzo  
łatwa w obsłudze:

```
<servlet>  
  <servlet-name>JerseyServlet</servlet-name>  
  <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>  
  <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
  <servlet-name>JerseyServlet</servlet-name>  
  <url-pattern>/*</url-pattern>  
</servlet-mapping>
```

# REST w javie czyli jersey-client

Jersey posiada podprojekt – bibliotekę kliencką do obsługi rest-ów

Biblioteka ta jest jedną z najprzyjemniejszych jakie miałem okazję ostatnio poznać – jest bardzo lekka w użyciu a jednocześnie ułatwia te rzeczy które ma ułatwiać przy pracy z http:

```
String s = Client.create().resource("http://localhost/hello").get(String.class);
```

# Tworzymy prosty webservice

Dodajemy do poma :

```
<dependency>  
  <groupId>com.sun.jersey</groupId>  
  <artifactId>jersey-server</artifactId>  
  <version>1.1.5</version>  
</dependency>
```

Do web.xml :

```
<servlet>  
  <servlet-name>JerseyServlet</servlet-name>  
  <servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>  
  <load-on-startup>1</load-on-startup>  
</servlet>  
<servlet-mapping>  
  <servlet-name>JerseyServlet</servlet-name>  
  <url-pattern>/*</url-pattern>  
</servlet-mapping>
```

# Tworzymy prosty webservice

## Tworzymy klasę helloWorld:

```
import javax.ws.rs.GET;  
import javax.ws.rs.Path;  
  
@Path("hello")  
public class helloWorld {  
  
    @GET  
    public String sayHello(){  
        return "hello";  
    }  
}
```

# Testujemy nasz webservice

<http://localhost:8080/helloworld/hello>

# Przekazywanie parametrów w ścieżce

```
import javax.ws.rs.GET;  
import javax.ws.rs.Path;  
import javax.ws.rs.PathParam;
```

```
@Path("hello")  
public class helloWorld {  
  
    @GET  
    @Path("/{name}")  
    public String sayHello(@PathParam("name") String fn){  
        return "hello "+fn;  
    }  
  
}
```

# Przekazywanie parametrów w parametrach wywołania

```
import javax.ws.rs.DefaultValue;  
import javax.ws.rs.GET;  
import javax.ws.rs.Path;  
import javax.ws.rs.QueryParam;
```

```
@Path("hello")  
public class helloWorld {
```

```
    @GET  
    public String sayHello(@DefaultValue("kowalski")  
                           @QueryParam("name") String n){  
        return "hello "+n;  
    }  
}
```

```
}
```

# Negocjacja formatu zwracanego

Różne aplikacje mogą oczekiwać danych w różnym formacie – przeglądarka www będzie najpewniej oczekiwała text/html , natomiast skrypt javascriptowy może chcieć application/json aplikacje w javie lub w .net z dużym prawdopodobieństwem będą oczekiwać xml-a

# Negocjacja formatu zwracanego

```
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("hello")
public class helloWorld {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String sayHello(){
        return "hello ";
    }

    @GET
    @Produces(MediaType.TEXT_HTML)
    public String sayHtmlHello(){
        return "<html><body><p>hello</p></body></html>";
    }
}
```

# Zwracanie własnych klas

## JAXB/JSON

jeżeli nasza klasa posiada anotacje JAXB – możemy ją bezpośrednio zwrócić jako `application/xml` lub `application/json`, konwersja na json jest automatyczna więc zawsze warto annotować obydwa typy jako typy zwracane.

# Annotacje JAXB

`@XmlRootElement` – annotację tę umieszczamy na klasie którą zwracamy.

W najprostszyc przypadkach to wszystko czego potrzebujemy.

`@XmlAttribute` – gdy chcemy by atrybut klasy był zapamiętany jako atrybut znacznika

`@XmlElement` – gdy chcemy by atrybut klasy był zapamiętany jako dziecko znacznika

`@XmlJavaTypeAdapter` – gdy prosty przypadek nie wystarcza – tj gdy natrafimy na typ którego jaxb nie umie zserializować (choćby `Map<String,String>` !)

# @XmlJavaTypeAdapter

```
public class StrStrMapAdaptor extends XmlAdapter<StrStrMyMap, Map<String,String>>{

    @Override
    public Map<String, String> unmarshal(StrStrMyMap v) throws Exception {
        Map<String, String> map = new HashMap<String,String>();
        for (Iterator<StrStrKeyVal> it = v.map.iterator(); it.hasNext();) {
            StrStrKeyVal strStr = it.next();
            map.put(strStr.key, strStr.value);
        }
        return map;
    }

    @Override
    public StrStrMyMap marshal(Map<String, String> v) throws Exception {
        StrStrMyMap map = new StrStrMyMap();
        for (Iterator<String> it = v.keySet().iterator(); it.hasNext();) {
            String key = it.next();
            StrStrKeyVal keyval = new StrStrKeyVal();
            keyval.key = key;
            keyval.value = v.get(key);
            map.map.add(keyval);
        }
        return map;
    }

    public static class StrStrMyMap{
        @XmlElement
        private List<StrStrKeyVal> map = new ArrayList<StrStrKeyVal>();
    }

    @XmlType
    @XmlAccessorType(XmlAccessType.NONE)
    public static class StrStrKeyVal{
        @XmlAttribute(name="key",required=true)
        private String key;
        @XmlElement
        private String value;
    }
}
```

# JAXB – ważne uwagi

JAXB nie radzi sobie z „final” - jest oczywiście w stanie je odczytać, ale już nie utworzyć później z tego obiekt !

Jeżeli przesyłamy obiekty modelowe i chcielibyśmy mieć w nich standardowe PropertyChangeListenery – wystarczy oznaczyć je jako transient !

# Pisanie klienta w plain java

Do pom.xml dodajemy :

```
<dependency>  
  <groupId>com.sun.jersey</groupId>  
  <artifactId>jersey-client</artifactId>  
  <version>1.1.5</version>  
</dependency>
```

A naszą klasą wyjściową jest :

```
com.sun.jersey.api.client.Client
```

I tak na prawdę puki mamy code completion –  
to wszystko co nam potrzebne :)

# Wyjątki

W prawie każdej aplikacji może się zdarzyć wyjątek – w restach mapuje się je na kody http.

```
public class NotFoundMap implements ExceptionMapper<NotFound> {  
  
    @Override  
    public Response toResponse(NotFound exception) {  
        return Response.status(Response.Status.NOT_FOUND)  
            .type(MediaType.TEXT_PLAIN)  
            .entity(exception.getMessage())  
            .build();  
    }  
}
```

# Wyjątki - klient

```
try{  
  
    //Kod łączenia się z serwerem  
  
} catch (UniformInterfaceException uniformInterfaceException) {  
    if (uniformInterfaceException.getResponse().getStatus() == 404){  
        throw new PlaceNotFound(placeId);  
    }  
    throw uniformInterfaceException;  
}
```