Rozproszone systemy internetowe

JAX-RS (RESTful web services)

cz.4

Subresources, HATEOAS, Filtry

HELP:

JAX-RS Tutorial by mkyong

Jersey-User guide-JAX-RS Application

Ćwiczenie 1. Definicja subresources

Utwórz subresource „comments” (komentarze utworzone do danego message)

Subresource comments powinien być dostępny przez URI:

GET http://localhost:8080/RestWS5/webresources/messages/1/comments

Help: Jersey-User guide-Resources and Sub-Resources

The second way @Path may be used is on methods **not annotated with resource method designators such as @GET or @POST**. Such methods are referred to as ***sub-resource locators***.

```
@Path("/item")

public class ItemResource {

    @Context UriInfo uriInfo;

    @Path("content")

    public ItemContentResource getItemContentResource() {

        return new ItemContentResource();

    }

    @GET

    @Produces("application/xml")

        public Item get() { ... }

    }

}

public class ItemContentResource {

    @GET
```

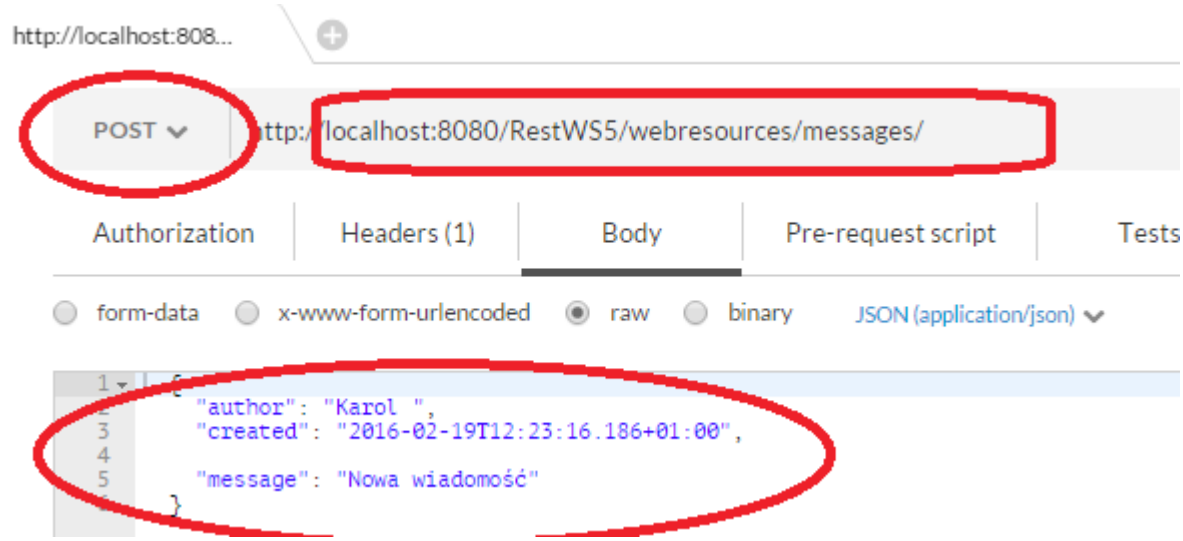
```
    public Response get() { ... }

    @PUT

    @Path("{version}")

    public void put(@PathParam("version") int version,

                    @Context HttpHeaders headers,

                    byte[] in) {

        ...

    }

}
```

Ćwiczenie 2. Zwrócenie adresu utworzonego zasobu w nagłówku odpowiedzi (Header-> Location)

Zmodyfikować medtodę tworzenia nowego message tak aby w nagłówku odpowiedzi był zwrócony aders nowego resource:

Location →http://localhost:8080/RestWS5/webresources/messages/5

Tworzenie nowego message JSON (Postman):



Response- w nagłówku odpowiedzi adres nowego zasobu

Builder Runner Import

http://localhost:808...

POST http://localhost:8080/RestWS5/webresources/messages/

Authorization | Headers (1) | Body | Pre-request script | Tests

form-data x-www-form-urlencoded raw binary JSON (application/json)

```
1 {
2   "author": "Karol ",
3   "created": "2016-02-19T12:23:16.186+01:00",
4
5   "message": "Nowa wiadomość"
6 }
```

Body Cookies Headers (6) Tests (0/0) Status 201 Created Time 337 ms

Content-Length → 103

Content-Type → application/json

Date → Tue, 23 Feb 2016 17:28:30 GMT

Location → http://localhost:8080/RestWS5/webresources/messages/5

Server → GlassFish Server Open Source Edition 4.1

X-Powered-By → Servlet/3.1 JSP/2.3 (GlassFish Server Open Source Edition 4.1 Java/Oracle Corporation/1.8)

Przykład kodu

```
49
50      @POST
51      @Consumes(MediaType.APPLICATION_JSON)
52      @Produces(MediaType.APPLICATION_JSON)
        public Response createMessage(Message message, @Context UriInfo uriInfo) {
54          Message newMessage = messageService.createMessage(message);
55          String newId = String.valueOf(newMessage.getId());
56          URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();
57          Response respose = Response.created(uri)
58                  .entity(newMessage)
59                  .build();
60          return respose;
61          // return messageService.createMessage(message);
62      }
63
```

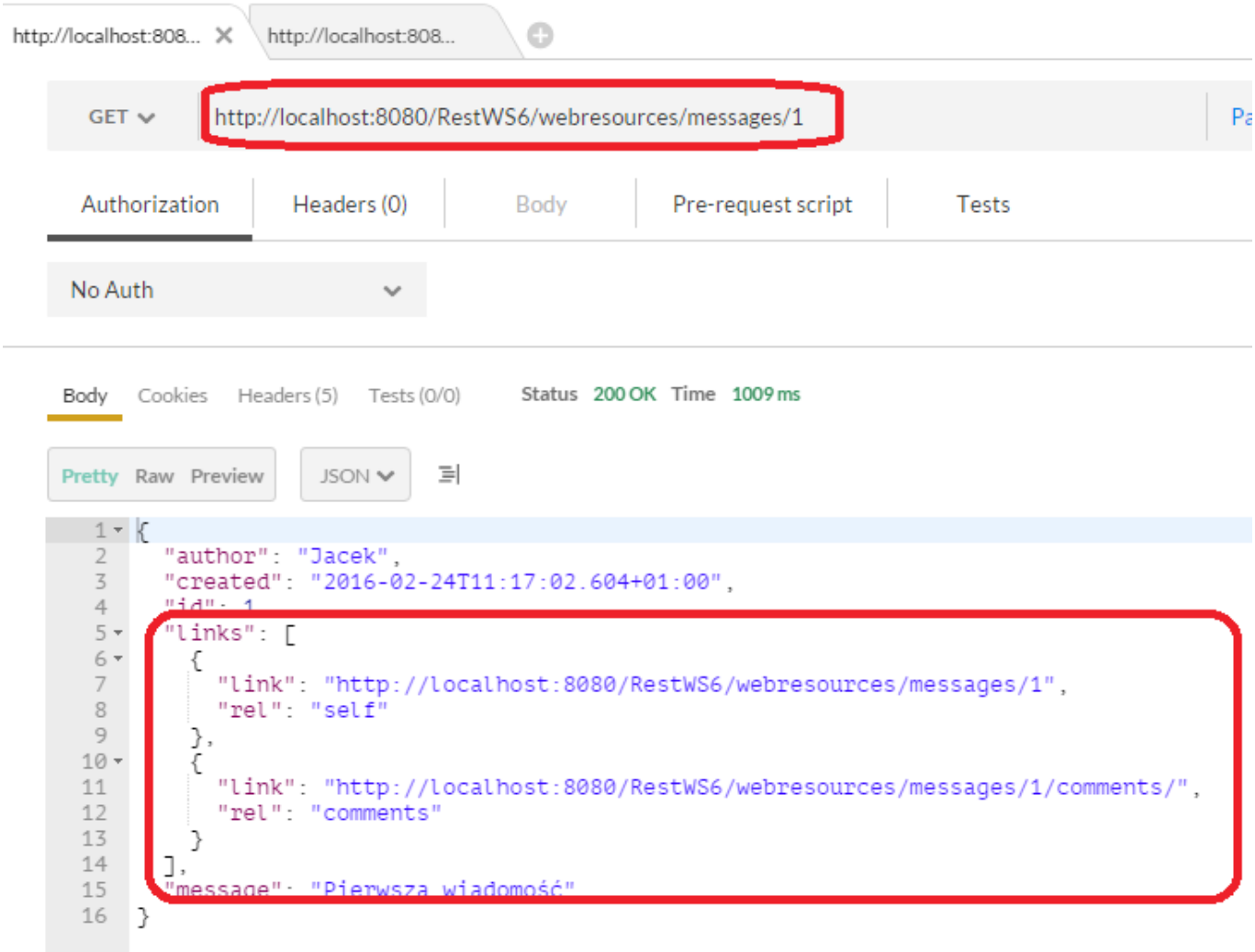
Ćwiczenie 3. Implementacja HATEOAS (Hypermedia as the Engine of Application State)

Most REST APIs have "help" pages that explain what the API URIs are and what operations are supported.

Let's say you receive a GET request from a client for a message ID. We return the message information in JSON or XML. But what you could also do is **send links to comment resource URIs.**

So, the web service is being super-helpful to the client by providing all these links in the response. Similar to hyperlinks in web sites. Whether the client wants to use it or not doesn't matter. But if they want it, it's there. **The client developer just picks up the value of the right URIs from a previous response and makes subsequent calls to those URIs.**

If you do this, you don't let the client programmer have to know and hard-code the URIs in order to interact with the resources and the application state. You basically let the hypertext you send in the response drive the client's interaction with the application state. So, you could say that hypertext, or hypermedia is being the driver or engine of application state. **Hypermedia as the Engine of Application State - HATEOAS.**



```java
43      @GET
44      @Path("/{messageId}")
45      @Produces(MediaType.APPLICATION_JSON)
        public Message getMessage(@PathParam("messageId") Long id, @Context UriInfo uriInfo) {
47          Message newMessage = messageService.getMessage(id);
48          String uri = uriInfo.getBaseUriBuilder()
49                  .path(MessageResource.class)
50                  .path(String.valueOf(newMessage.getId()))
51                  .build()
52                  .toString();
53          newMessage.addLink(uri, "self");
54
55          String uri2 = uriInfo.getBaseUriBuilder()
56                  .path(MessageResource.class)
57                  .path(MessageResource.class,"getComments" )
58                  .path(CommentResource.class)
59                  .resolveTemplate("messageId", newMessage.getId())
60                  .build()
61                  .toString();
62          newMessage.addLink(uri2, "comments");
63
64          return newMessage;
```

```java
public class Message {

private long id;

private String message;

private Date created;

private String author;

private List<Link> links = new ArrayList<Link>();
```

Ćwiczenie 4. Singleton

Jaka będzie różnica gdy oznaczysz klasę serwisu REST adnotacją **@Singleton?**

Sprawdź w przykładzie.

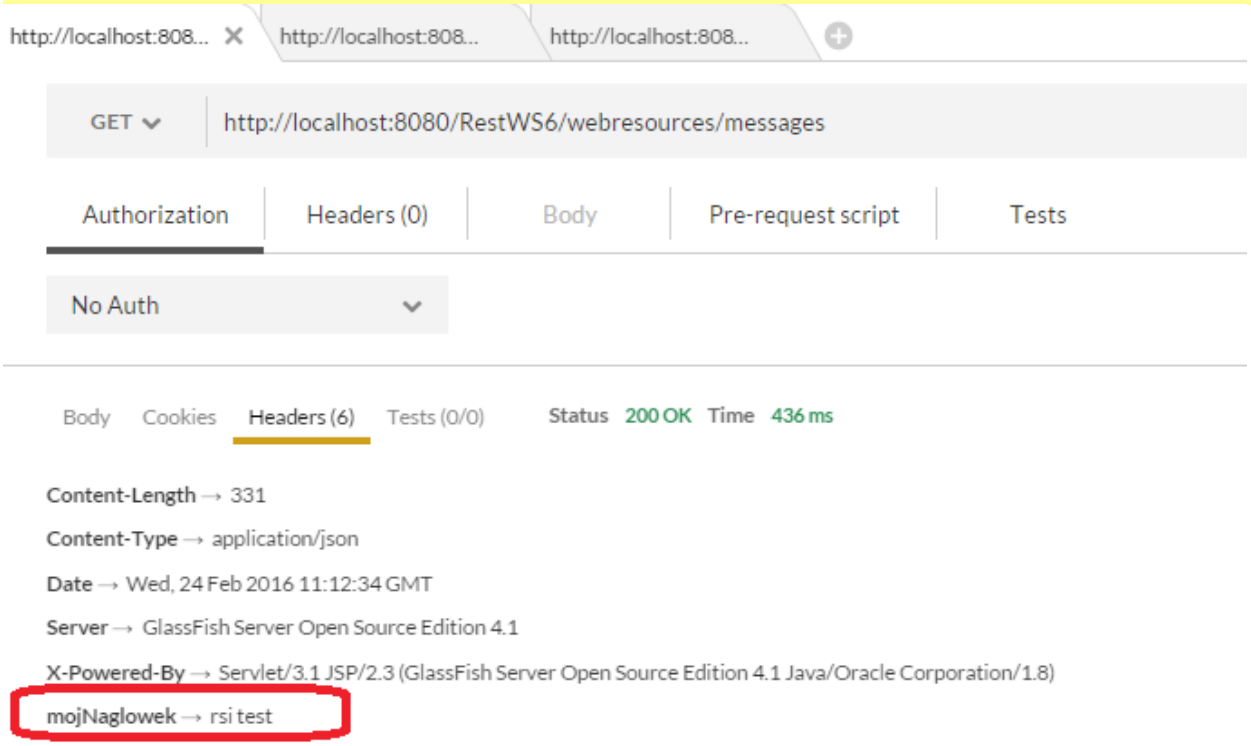
Ćwiczenie 5. Filtry

Możliwość modyfikacji odpowiedzi (Response) zwracanej przez REST API

Help: Jersey- User guide- Filters and Interceptors

Zaimplementuj ResponseFilter:

```
package filter;

import java.io.IOException;

import javax.ws.rs.container.ContainerRequestContext;

import javax.ws.rs.container.ContainerResponseContext;

import javax.ws.rs.container.ContainerResponseFilter;

import javax.ws.rs.ext.Provider;

@Provider

public class MyResponseFilter implements ContainerResponseFilter {

@Override

public void filter(ContainerRequestContext requestContext, ContainerResponseContext
responseContext) throws IOException {

responseContext.getHeaders().add("mojNaglowek", "rsi test");

}

}
```



Rezultat:

Implementacja Request i Responce filter w jednej klasie

```
@Provider
public class LogingFilter implements ContainerRequestFilter , ContainerResponseFilter{

    @Override
    public void filter(ContainerRequestContext requestContext) throws IOException {
        System.out.println("Request Filter:");
        System.out.println("Headers:" + requestContext.getHeaders());
    }

    @Override
    public void filter(ContainerRequestContext requestContext, ContainerResponseContext response
    System.out.println("Response Filter:");
     System.out.println("Headers:" + responseContext.getHeaders());
    }
```

```
Info:   Registering the Jersey servlet application, named service.ApplicationConfig, at the servlet mapping /web
Info:   Initiating Jersey application, version Jersey: 2.10.4 2014-08-08 15:09:00...
Info:   Loading application [RestWS6] at [/RestWS6]
Info:   RestWS6 was successfully deployed in 231 milliseconds.
Info:   Tworzenie Db
Info:   Request Filter:
Info:   Headers:{host=[localhost:8080], connection=[keep-alive], cache-control=[no-cache], user-agent=[Mozilla/5
Info:   Response Filter:
Info:   Headers:{Content-Type=[application/json]}
```

Ćwiczenie 6. Filtry – API Authentification

Zaimplementuj **filtr** który odczyta usera i password przesłane metodą **Basic Auth**