Lecture 3: REST Web Service (with Jersey)

TIES4560 SOA and Cloud Computing
Autumn 2023

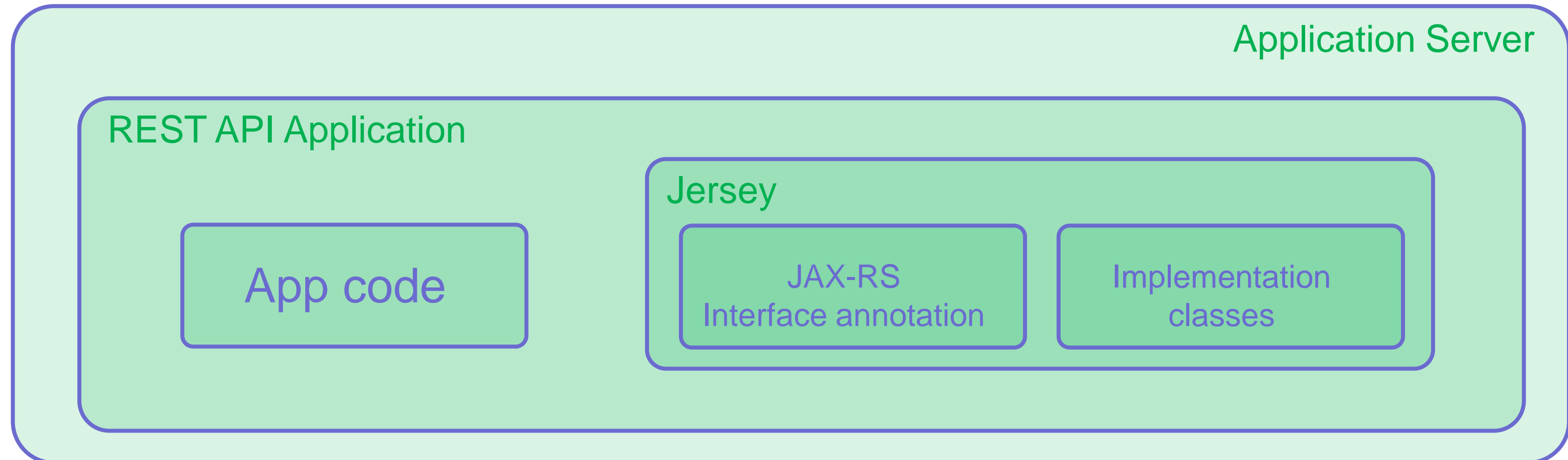
Khriyenko Oleksiy

REST Web Services

JAX-RS

JAX-RS *is an API for RESTful Web Services. JAX-RS contains Interfaces, therefore, to build an App we need actual implementation of them.*

There are many implementation libraries of the API (e.g. ***Rest Jersey***, ***Restlet***, ***RESTEasy****, etc.).*



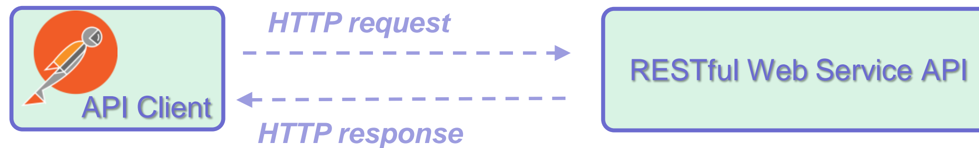
… Jersey is developed by people who wrote specification for JAX-RS (javax.ws.rs.)*

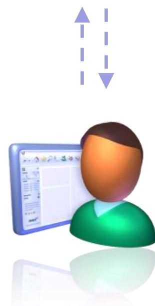
… learn one implementation and you will almost know all of them!!!

Related tutorials:

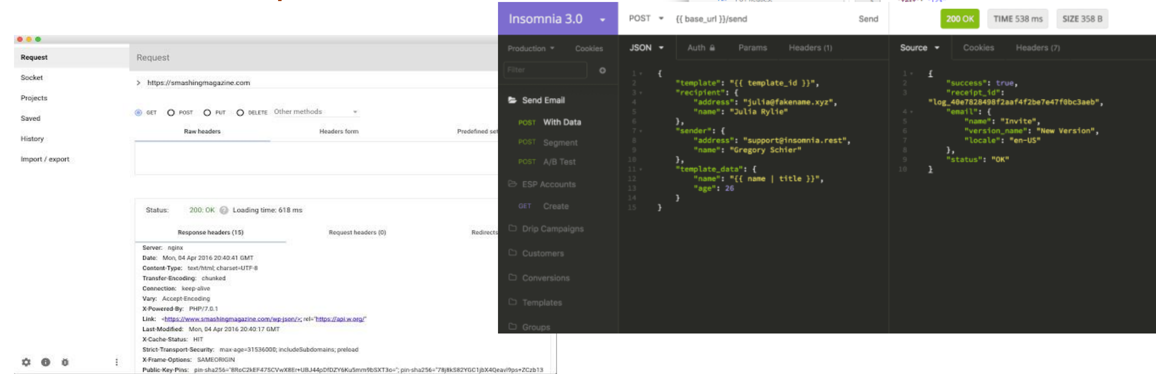
- *http://docs.oracle.com/javaee/6/tutorial/doc/giepu.html*
- *http://www.tutorialspoint.com/restful/restful_jax_rs.htm*
- *http://www.mkyong.com/tutorials/jax-rs-tutorials/*
- *http://crunchify.com/how-to-build-restful-service-with-java-using-jax-rs-and-jersey/*

REST Web Services





REST API Client *is the web developers helper program to create and test custom HTTP requests.*



Chrome

Postman: https://chrome.google.com/webstore/detail/postman/fhbjgbiflinjbdggehcddcbncdddomop?hl=en

Advanced REST client: https://chrome.google.com/webstore/detail/advanced-rest-client/hgmloofddffdnphfgcellkdfbfbjeloo

DHC: https://chrome.google.com/webstore/detail/dhc-rest-client/aejoelaoggembcahagimdiliamlcdmfm?hl=en

Firefox

Firefox add-on: https://addons.mozilla.org/en-US/firefox/addon/restclient/

Insomnia *is a cross-platform application for organizing, running, and debugging HTTP requests (https://insomnia.rest/).*

REST Web Services

Jersey RESTful Web Services framework *is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339 & JSR 370) Reference Implementation. Jersey provides it's own API that extend the JAX-RS toolkit with additional features and utilities to further simplify RESTful service and client development.* (*https://eclipse-ee4j.github.io/jersey/*)

Jersey is distributed mainly via ***Maven***

Archetype Group Id: ***org.glassfish.jersey.archetypes***

Archetype Artifact Id: ***jersey-quickstart-webapp***

Archetype Version:

the latest published release of Jakarta EE 9 Jersey is 3.0.8 (versions 3.x requires Tomcat v10)

for Tomcat v9, downgrade Jersey version to 2.x (the latest stable release of Jersey is 2.37)

Modify ***web.xml*** *file to map the Servlet that is provided by Jersey package with corresponding URL pattern.*

So, all the REST API requests will be preconfigured to go to the URL

`<port>/<artifactId>/webapi/`

```
…
<servlet-mapping>
     <servlet-name>Jersey Web Application</servlet-name>
     <url-pattern>/webapi/*</url-pattern>
</servlet-mapping>
…
```

… from now, just ***start to create resources***
by providing corresponding annotations to the classes!!!

Related tutorials:

- *https://mvnrepository.com/artifact/org.glassfish.jersey.core/jersey-client*
- *https://stackoverflow.com/questions/62927426/getting-java-lang-classnotfoundexception-jakarta-servlet-filter-on-maven-jersey*

REST Web Services

Application class in Jersey…

This is another way to configure REST API Application as an alternative to Servlet configuration in web.xml file.

- *Use **@ApplicationPath** annotation to specify the URL your API is mapped to (e.g. **webapi**)*
- *By default, Jersey checks all the resources with **@Path** annotation in your application and add them to the list to be managed. Alternatively, you may directly specify classes that have to be considered by Jersey via implementation (overwriting a default implementation) of the **getClasses()** method. In this way you may exclude some classes from the consideration…*

```
…
@ApplicationPath("webapi")
public class MyRESTApp extends Application{

   public Set<Class<?>> getClasses(){
      return new HashSet<Class<?>>();    //this returns empty set of classes (add there ...)
}
```

*… from now, just **start to create resources** by providing corresponding annotations to the classes!!!*

Related tutorials:

- *https://eclipse-ee4j.github.io/jersey.github.io/documentation/latest/index.html*

REST Web Services

@Path annotation's value is a relative URI path.

@GET, @PUT, @POST, @DELETE and @HEAD are resource method designator annotations defined by JAX-RS and which correspond to the HTTP methods.

@Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client (text/plain, application/xml, application/json).

```
…
@Path("/publications")
public class PublicationResource {
            @GET
            @Produces("text/plain")
//          @Produces({"application/xml", "application/json"})
//          @Produces(value={MediaType.APPLICATION_JSON, MediaType.TEXT_XML})
            public String getPublications(){
                        return "Publications...";
            }
}
```

*To return JSON Response we need corresponding convertor to JSON format. For this purpose, you may uncomment corresponding dependency reserved in **pom.xml** file.*

```
<!-- uncomment this to get JSON support -->
<dependency>
    <groupId>org.glassfish.jersey.media</groupId>
    <artifactId>jersey-media-json-binding</artifactId>
</dependency>
```

REST Web Services

@XmlRootElement *annotation specifies a root element for **JAXB**, which is used by Jersey to generate XML schema from Java objects.*

*Java Architecture for XML Binding (**JAXB**) provides a fast and convenient way to bind XML schemas and Java representations, making it easy for Java developers to incorporate XML data and processing functions in Java applications. As part of this process, JAXB provides methods for reading XML instance documents into Java content trees, and then writing Java content trees back into XML instance documents.* (https://docs.oracle.com/javase/tutorial/jaxb/intro/)

```
…
@Path("/publications")
public class PublicationResource {
   PublicationService publicationService = new
PublicationService();

   @GET
   @Produces(MediaType.APPLICATION_XML)
   public List<Publication> getPublications(){
      return publicationService.getAllPublications();
   }
}
```

```
…
@XmlRootElement
public class Publication {

   private long id;
   private String title;
   private Date published;
   private String mainAuthor;

   public Publication(){}
…
}
```

Be sure that you have all getters and setters for class variables to generate proper XML(JSON) output…


```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <publications>
  - <publication>
      <id>1</id>
      <mainAuthor>me</mainAuthor>
      <published>2016-09-16T19:41:52.375+03:00</published>
      <title>Conference paper 1</title>
    </publication>
  - <publication>
      <id>2</id>
      <mainAuthor>me</mainAuthor>
      <published>2016-09-16T19:41:52.375+03:00</published>
      <title>Juornal paper 1</title>
    </publication>
  </publications>
```

REST Web Services

/publications/{publicationId}/

*- to get nested URI (nested path) use the same **@Path** annotation for a class method.*

- *since we cannot hardcode the path of a resource accessed by **Id**, we use **{variable name}** instead of concrete part of the URL.*
- *to get an access to the variable use **@PathParam("variable name")** annotation for method's argument.*

```
…
@Path("/publications")
public class PublicationResource {
    PublicationService publicationService = new PublicationService();
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public List<Publication> getPublications(){
        return publicationService.getAllPublications();
    }
    @GET
    @Path("/{publicationId}")
    @Produces(MediaType.APPLICATION_JSON)
    public Publication getPublication(@PathParam("publicationId") long id){
        Publication publication = publicationService.getPublication(id);
        return publication;
    }
}
```

- *the **variable** in @Path annotation may be customized by specifying a different regular expression after the variable name (default regular expression is [^/]+?). For example, if a user name must begin with one uppercase or lowercase letter and zero or more alphanumeric characters and the underscore character. If a user name does not match that template, a 404 (Not Found) response will be sent to the client.*

```
@Path("users/{username: [a-zA-Z][a-zA-Z_0-9]*}")
```

REST Web Services

*Use **@POST** annotation to identify method that consumes HTTP POST request and creates new resource based on resource sent within the request body.*

***@Consumes** annotation is used to specify the MIME media types of representations a resource can consume from the client (text/plain, application/xml, application/json).*

```
…
@Path("/publications")
@Produces(MediaType.APPLICATION_JSON)
public class PublicationResource {
    PublicationService publicationService = new PublicationService();
    @GET
    @Produces(MediaType.APPLICATION_XML)
    public List<Publication> getPublications(){
        return publicationService.getAllPublications();
    }
    @GET
    @Path("/{publicationId}")
    public Publication getPublication(@PathParam("publicationId") long id){
        Publication publication = publicationService.getPublication(id);
        return publication;
    }
    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Publication addPublication (Publication publication){
        return publicationService.addPublication(publication);
    }
}
```

REST Web Services

*Use **@PUT** annotation to identify method that consumes HTTP PUT request and updates specified resource with resource sent within the request body.*

*Use **@DELETE** annotation to identify method that consumes HTTP DELETE request and deletes specified resource.*

```
…
@Path("/publications")
@Produces(MediaType.APPLICATION_JSON)
public class PublicationResource {
   PublicationService publicationService = new PublicationService();
   …
   @PUT
   @Path("/{publicationId}")
   @Consumes(MediaType.APPLICATION_JSON)
   public Publication updatePublication(@PathParam("publicationId") long id, Publication
                                                                        publication){
      publication.setId(id);
      return publicationService.updatePublication(publication);
   }
   @DELETE
   @Path("/{publicationId}")
   public void deletePublication (@PathParam("publicationId") long id){
      publicationService.removePublication (id);
   }
}
```

REST Web Services

Filtering and Pagination *require use of* ***query parameters*** *of the request.*

```
…
public List<Publication> getAllPublicationsForYear(int year){
   List<Publication> publicationsForYear = new ArrayList<>();
   Calendar cal = Calendar.getInstance();
   for (Publication publication : publications.values()){
      cal.setTime(publication.getPublished());
      if (cal.get(Calendar.YEAR)== year){ publicationsForYear.add(publication); }
   }
   return publicationsForYear;
}
public List<Publication> getAllPublicationsPaginated(int start, int size){
   ArrayList<Publication> list = new ArrayList<Publication>(publications.values());
   if (start + size > list.size()) return new ArrayList<Publication>();
   return list.subList(start,start+size) ;
}
…
```

/publications?year=2018

and

/publications?start=5&size=10

Use ***@QueryParam("parameter name")*** *annotation for method's argument to get a value of the parameter .*

```
…
@GET
public List<Publication> getPublications(@QueryParam("year") int year,
                                         @QueryParam("start") int start,
                                         @QueryParam("size") int size){
   if(year > 0){ return publicationService.getAllPublicationsForYear(year); }
   if(start >= 0 && size > 0){
      return publicationService.getAllPublicationsPaginated(start, size);
   }
   return publicationService.getAllPublications();
}
…
```

REST Web Services

Param annotations:

- ***@MatrixParam("parameter name")*** *annotation is similar to* ***@QueryParam*** *and used for the cases when parameters are separated by (;) in the request.* /publications;year=2016;size=5
- ***@HeaderParam("parameter name")*** *annotation is used to access an extra metadata in a form of custom header values of the request.*
- ***@CookieParam("parameter name")*** *annotation to access values of the cookie's names.*
- ***@FormParam("parameter name")*** *annotation to access "key:value" pairs in HTML Form submissions.*

In the mentioned cases you suppose to know parameter names in advance. If you do not have such opportunity, you are able to get them and other useful metadata from the ***Context*** *of the request using* ***@Context*** *annotation and corresponding components:*

- ***UriInfo*** *provides both static and dynamic, per-request information, about the components of a request URI (e.g. absolute path .getAbsolutePath(), base URI .getBaseUri(), query parameters .getQueryParameters(), etc.).*
- ***HttpHeaders*** *provides access to request header information either in map form or via strongly typed convenience methods. (e.g. names of all the headers .getRequestHeaders(), cookies .getCookies(), date .getDate(), accepted Media Types .getAcceptedMediaTypes(), etc.).*

```
public String getParamsUsingContext(@Context UriInfo uriInfo,
                                    @Context HttpHeaders headers){

   String path = uriInfo.getAbsolutePath().toString();
   String cookies = headers.getCookies().toString();
   return "Path: "+path+"; Cookies - "+cookies;
}
```

There are also other components as: ***SecurityContext, ResourceContext, Request, Configuration, Application, Providers****.*

REST Web Services

@BeanParam *annotation that may be used to inject custom JAX-RS "parameter aggregator" value object into a resource class field, property or resource method parameter.*

```
public class PublicationFilterBean {

   private @QueryParam("year") int year;
   private @QueryParam("start") int start;
   private @QueryParam("size") int size;

   public int getYear() {return year;}
   public void setYear(int year) {this.year = year;}
   public int getStart() {return start;}
   public void setStart(int start) {this.start = start;}
   public int getSize() {return size;}
   public void setSize(int size) {this.size = size;}
}
```

You may aggregate various annotations under one class that will contain them. It will definitely make your code more readable.

Also, use this approach especially in case you are not sure which concrete parameter is used.

```
…
@GET
public List<Publication> getPublications(@BeanParam PublicationFilterBean fBean){
   if(fBean.getYear() > 0){
     return publicationService.getAllPublicationsForYear(fBean.getYear());
   }
   if(fBean.getStart() >= 0 && fBean.getSize() > 0){
     return publicationService.getAllPublicationsPaginated(fBean.getStart(), fBean.getSize());
   }
   return publicationService.getAllPublications();
}
…
```

REST Web Services

@...Param *annotations on the level of a class variables vs. class method's attributes…*

It allows you to use ***...Param*** *values in all the methods of the class.*

```
…
@Path("/publications")
@Produces(MediaType.APPLICATION_JSON)
public class PublicationResource {
   @PathParam("publicationId") private long pubId;
   @QueryParam("year") private int pubYear;
   @QueryParam("start") private int pubStart;
   @QueryParam("size") private int pubSize;

   PublicationService publicationService = new PublicationService();
   …
   @GET
   public List<Publication> getPublications(@QueryParam("year") int year,
                                            @QueryParam("start") int start,
                                            @QueryParam("size") int size){
      if(year > 0){ return publicationService.getAllPublicationsForYear(year); }
      if(start >= 0 && size > 0){
         return publicationService.getAllPublicationsPaginated(start, size);
      }
      return publicationService.getAllPublications();
   }
   @DELETE
   @Path("/{publicationId}")
   public void deletePublication (@PathParam("publicationId") long id){
      publicationService.removePublication (id);
   }
}
```

REST Web Services

Param Converter for a custom type conversion.

- *Jersey has a set of default **ParamConvertor**s to deal with basic types to convert from a String.*
- *To manage conversion to custom type we need to implement **ParamConvertor** interface as well as **ParamConvertorProvider** that will be registered to Jersey.*

```
@Path("call/{deadline}")
public class CallForPaperResource {
   @GET
   @Produces(MediaType.TEXT_PLAIN)
   public String getRequestedCall(@PathParam("deadline") MyDate date){return "Date:"+date.toString();}
}
```

```
@Provider // the annotation preregisters our Provider for JAX-RS to be used
public class MyDateConverterProvider implements ParamConverterProvider {
    @Override
    public <T> ParamConverter<T> getConverter(final Class<T> rawType, Type genericType, Annotation[]
                                                                                  annotations) {
        if(rawType.getName().equals(MyDate.class.getName())){
            return new ParamConverter<T>(){
                @Override
                public T fromString(String value){
                    MyDate myDate = null;
                    if("submission".equalsIgnoreCase(value)){ myDate = new MyDate(31,10,2023); }
                    if("notification".equalsIgnoreCase(value)){ myDate = new MyDate(30,11,2023); }
                    if("cameraReady".equalsIgnoreCase(value)){ myDate = new MyDate(31,12,2023); }
                    return rawType.cast(myDate);
                }
                @Override
                public String toString(T bean) { if(bean == null){return null;} return bean.toString(); }
            };
        }return null;
    }
}
```

You may use such conversion when you do not want to implement the same conversion logic in different places of your application.

REST Web Services

per-Request *and* ***Singleton Resources*** *in JAX-RS*

- *By default, a class that represents a resource is initialized (new instance is created) every time when resource is requested (**per-Request**).*
- *If you need to keep some data, keep the data in a separate object (instance of other class) or make the resource **Singleton** with corresponding annotation.*

```
…
@Path("secured")
@Produces(value={MediaType.TEXT_PLAIN, MediaType.APPLICATION_JSON})
@Singleton
public class SecuredResource {

   private int counter;

   @GET
   @Path("degrees")
   public String securedMethod() {
      counter++;
      return "This secured API is called " + counter + " time(s)...";
   }
}
```

- *It is **not allowed** to use **…Param annotation** on a level of class variables (only on a level of class method's attributes).*

REST Web Services

/publications/{publicationId}/comments

/publications/{publicationId}/comments/{commentId}

When we implement API for ***nested resources****, we may do it in the same class of the root resource (but it might not be so convenient).*

```
@GET
@Path("/{publicationId}/comments")
public List<Comment> getComments(@PathParam("publicationId") long publicationId){
   return commentService.getAllComments(publicationId);
}
```

So, it is much more reasonable to do this for new (nested) resource in a separate class and make a "reference" (delegate further actions) to it from the class of our root (parent) resource…

```
@Path("/{publicationId}/comments")
public CommentResource getCommentResource(){
   return new CommentResource(); }
```

```
@Path("/")
@Consumes(MediaType.APPLICATION_JSON)
@Produces(MediaType.APPLICATION_JSON)
public class CommentResource {
   private CommentService commentService = new CommentService();
   @GET
   public List<Comment> getComments(@PathParam("publicationId") long publicationId){
      return commentService.getAllComments(publicationId); }
   @GET
   @Path("/{commnetId}")
   public Comment getComment(@PathParam("publicationId") long publicationId,
                                                        @PathParam("commentId") long commentId){
      return commentService.getComment(publicationId, commentId); }
   …
}
```

REST Web Services

Response *might return not only a content in the body. Additionally, service may return extra metadata (status codes and various headers).*

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Publication addPublication (Publication publication){
    return publicationService.addPublication(publication);
}
```

- returns only Publication resource in JSON format inside a response body.

Use a response builder to enrich response with metadata.

```
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response addPublication (Publication publication){
    Publication newPub = publicationService.addPublication(publication);
    return Response.status(Status.CREATED)
                   .header("Location", … )
                   .entity(newPub)
                   .build();
}
```

- returns status code 201-Created and URI of newly created resource in the location header in addition to Publication resource in JSON format inside a response body.

(Similarly, you can add cookie, encoding and various headers as well).

You may use quick shortcut to specify both "created" status code and location of created resource… Use ***@Context UriInfo*** *and* ***URI builder*** *to simplify construction of resource URI .*

```
…
public Response addPublication (Publication publication, @Context UriInfo uriInfo){
    Publication newPub = publicationService.addPublication(publication);
    String newId = String.valueOf(newPub.getId());
    URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();
    return Response.created(uri)
                   .entity(newPub).build();
}
```

REST Web Services

HATEOAS *(Hypermedia As The Engine Of Application State)*

*To make Response **navigable** we need to add links to related resources into Response.*

*We have to extend a model of our Publication object with new variable Link that contains **href** and **rel** variables inside.*

```
{  "id" : "10",
   "title"  : "Publication 123",
   "mainAuthor" : "me",
   "published" : "…",
   "co-Authors" : […],
   "links" : [  {  "href" : "/publications/10",
                   "rel" : "self"  },
                {  "href" : "/publications/10/comments",
                   "rel" : "comments"  },
                {  "href" : "/profiles/3",
                   "rel" : "mainAuthorProfile"  }
            ]
}
```

```
…
public class Link {
   private String link;
   private String rel;
// below generate getters and setters for all the variable of the class…
}
```

```
…
@XmlRootElement
public class Publication {
   private long id;
   private String title;
   private Date published;
   private String mainAuthor;
   private List<Author> coAuthors = new ArrayList<>();
   private List<Link> links = new ArrayList<>();

   public Publication(){}
   public void addLink(String url, String rel){
      Link link = new Link();
      link.setLink(url);
      link.setRel(rel);
      links.add(link);
   }
…
}
```

REST Web Services

HATEOAS …

Add the links to Publication resource…

```
…
@POST
@Consumes(MediaType.APPLICATION_JSON)
public Response addPublication(Publication publication, @Context UriInfo uriInfo){
   Publication newPublication = publicationService.addPublication(publication);
   String uri = uriInfo.getBaseUriBuilder()        http://localhost:8080/MyRESTws/webapi/
                .path(PublicationResource.class)                              /publications
                .path(Long.toString(publication.getId()))                   /{publicationId}
                .build()
                .toString();
   newPublication.addLink(uri,"self");
   …
// do similarly for "comments" link…
   uri = uriInfo.getBaseUriBuilder()                http://localhost:8080/MyRESTws/webapi/
         .path(PublicationResource.class)                                     /publications
         .path(PublicationResource.class, "getCommentResource")  /{publicationId}/comments
         .resolveTemplate("publicationId", publication.getId())
         .build()
         .toString();
   newPublication.addLink(uri,"comments");
   String newId = String.valueOf(newPublication.getId());
   URI uri = uriInfo.getAbsolutePathBuilder().path(newId).build();
   return Response.created(uri)
                  .entity(newPublication)
                  .build();
}
```

nested resource

REST Web Services

Handle Exceptions ...

Create own classes for exceptions (e.g. DataNotFoundException) that extend RuntimeException.
(add generated serialVersionUID since it is required by any RuntimeException)

```
public class DataNotFoundException extends RuntimeException{
   private static final long serialVersionUID = -6672553621676928689L;
   public DataNotFoundException(String message){
      super(message);
   }
}
```

- constructor simply takes a message and pass it to the parent.

```
public Publication getPublication(long id){
   Publication publication = publications.get(id);
   if(publication == null){ throw new DataNotFoundException("Publication
                                             with id "+id+" not found"); }
   return publication;
}
```

- method throws an exception in case there is no publication with requested Id.

The service throws our exception to the resource handler, resource handler throws it further to the framework. To actually handle an exception thrown by above method, we have to make framework able to catch it and return a JSON response:

- *Create a JSON Response*
- *Map an exception to the JSON Response*

REST Web Services

Handle Exceptions …

Create JSON Response by creating a new class ErrorMessage that will represent a JSON object…

```
@XmlRootElement
public class ErrorMessage {
   private String errorMessage;
   private int errorCode;          //own custom error code
   private String documentation;   //link to documentation regarding an error and it's resolving
   public ErrorMessage(){}
   public ErrorMessage(String errorMessage, int errorCode, String documentation) {
      super();
      this.errorMessage = errorMessage;
      this.errorCode = errorCode;
      this.documentation = documentation;
   }
// below generate getters and setters for all the variable of the class…
}
```

Map the exception to our Response using ExceptionMapper class.

```
@Provider       // the annotation preregisters our Mapper for JAX-RS to be used
public class DataNotFoundExceptionMapper implements ExceptionMapper<DataNotFoundException>{
   @Override
   public Response toResponse(DataNotFoundException ex) {
      ErrorMessage errorMessage = new ErrorMessage(ex.getMessage(),404,"http://myDocs.org");
      return Response.status(Status.NOT_FOUND)
                     .entity(errorMessage)
                     .build();
   }
}
```

REST Web Services

Handle Exceptions …

You may create a bunch of other Mappers for different Exceptions you would like to handle.
*Only thing you need is to decide which **status code** you would like to use for that case…*

*It is always good to have a **Generic Exception Mapper** that will handle whatever is thrown if there is no explicit mapper for it (using **Throwable** as a synonym to Catch All).*

```
@Provider
public class GenericExceptionMapper implements ExceptionMapper<Throwable>{
   @Override
   public Response toResponse(Throwable ex) {
      ErrorMessage errorMessage = new ErrorMessage(ex.getMessage(),500,"http://myDocs.org");
      return Response.status(Status.INTERNAL_SERVER_ERROR)
                     .entity(errorMessage)
                     .build();
   }
}
```

REST Web Services

Handle Exceptions …

JAX-RS has its own set of exceptions that are mapped to a status of a response. The parent class of those exceptions is WebApplicationException.

Do not forget to disable a Generic Exception Mapper since it will interfere with WebApplicationException handler

Since WebApplicationException is a parent class, there are a lot of classes that are inherited from it and provide custom responses and statuses (e.g. NotFoundException, InternalServerErrorException , etc.).

Check JAX-RS javadoc for more detailed descriptions of those subclasses

```
public Comment getComment(long publicationId, long commentId){
   Publication publication = publications.get(publicationId);
   ErrorMessage errorMessage = new ErrorMessage("Not found...", 404, "http://myDocs.org");
   Response response = Response.status(Status.NOT_FOUND)
                               .entity(errorMessage)
                               .build();
   if(publication == null){
      throw new WebApplicationException(response);
   }
   Map<Long, Comment> comments = publications.get(publicationId).getComments();
   Comment comment = comments.get(commentId);
   if(comment == null){
      throw new NotFoundException(response);
   }
   return  comment;
}
```

REST Web Services

some HINTs …

- *Hide some of resource properties from XML or JSON conversion (useful for nested resources)…*

```
…
@XmlRootElement
public class Publication {
   private long id;
   private String title;
   private Date published;
   private String mainAuthor;
   private List<Author> coAuthors = new ArrayList<>();
   private Map<Long, Comment> comments = new HashMap<>();
   private List<Link> links = new ArrayList<>();

   public Publication(){}

   @XmlTransient
   public Map<Long, Comment> getComments() {
      return comments;
   }
…
}
```

Annotate fields that we do not want to be included in XML or JSON output with @XMLTransient.

- *Use content negotiation feature… Playing around with various combinations of @Consumes and @Produces annotations for the methods, you may provide different implementations and logics for service consumers who specify content type of sent request body and type of acceptable response using corresponding values in the request header (Content-Type header and Accept header respectively).*