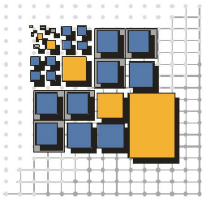

Otto-Friedrich-University Bamberg

Distributed Systems Group

Service Oriented Architecture (SOA)

Assignment 1

Introduction and Description
(Summer term 2020)

Preface

REST as an architectural style dominates modern web application development. Beginning in 2000 (or even before in the mid 90s), HTTP 1.1 and REST changed the way software developers design their systems. A quote may show why REST prevailed over other architectural and technological fields:

> "I think something dramatically simpler than WSDL could get the job done most of the time. We know the hard things are possible, we just have to make the easy things easy."
> (Norman Walsh[1])

REST is probably that successful because it is easy to implement (to some maturity level - we think maturity level 2 as many others). It does not have a overwhelming specification with a lot of edge cases and caveats.

It's an architectural style, usually combined with HTTP and its verbs to get it in production. During this intro we will shed some light on how to implement a REST API in Java without any framework besides the reference implementation Jersey. We add a documentation for this via Swagger and try to give the used data types some kind of semantics.

The following list is a recommendation which papers are worth reading and where you find some conceptual help when designing your API:

- **[Fie00]**: PhD thesis. One of the most cited REST sources as Fielding established the term Representational State Transfer (REST) in his PhD thesis. Chapter 5 is especially relevant but rather theoretical/conceptual. The reference is for completeness, but rather spend your time on reading one of the other sources.
- **[Pau14]**: Good overview of principles, patterns, technologies and maturity models. A running example, where common pitfalls are stated and which targets the least used HATEOAS principle of REST APIs (highly recommended).
- **[RBD+16]**: Study of the Italian HTTP traffic of a single day. Also uses the maturity model of Richardson to assess the quality of REST APIs. "More surprising is that the data format `application/json` is already on the forth position, while `text/xml` is only on the ninth position." (page 29) (the study is from 2016!)
  "[...] the dataset contains 1134 different media type declarations." (page 31)

[1] `https://norman.walsh.name/2005/02/24/wsdl`

Contents

1 What is a Web API?

1.1 General Considerations

To get a first idea, we split the phrase into its components. *Web* means the world wide web, where resources are identified via URLs. You can use these resources by standardized protocols and frameworks (i.e. HTTP 1.1/2.0). *API* (Application Programming Interface) describes (in our case the provider) which methods/operations your interface should implement from an implementation perspective. From a user perspective (consumer), it describes which methods/operations are offered and (hopefully documented) how these operations can be used.

Arnaud Lauret describes a *Web API* in his book as follows: "[A Web API] looks like a software version of the LEGO bricks system; [...] Each software brick can be used at the same time by many others. [...] Each software brick can run anywhere on its own as long as it's connected to a network in order to be accessible via its API", [Lau19, 7].

So let's play implementation LEGO and build our first API which is as intuitive as LEGO bricks are.

1.2 REST Principles

REST (REpresentational State Transfer) is an architectural style to design Web APIs. It was introduced in Roy Fieldings PhD thesis [Fie00] and became a de facto standard for APIs consumed via the internet. Since Fielding also contributed to the HTTP 1.1 standard, REST relies on many WWW principles. It's important to note that REST and its ideas are not bound to any implementation technology. In the following paragraphs, the principles of this architectural style are discussed.

1.2.1 Identification of Resources

"The key abstraction of information in REST is a resource. Any information that can be named can be a resource" [Fie00, 88]. A resource is a mapping to a set of resource identifiers and/or resource representations for a given point in time. "The only thing that is required to be static for a resource is the semantics of the mapping, since the semantics is what distinguishes one resource from another." [Fie00, 89].

Mappings can be dynamic or static, but the semantics of the mapping must be static and must not change over time. For example, a resource could be `"the oldest human alive"`, where the mapping is dynamic because its value, the name (identifier) and the actual person (representation), changes every time the currently oldest person dies. The resource `"the father of John Doe"`, however, would be a static mapping, because the value will always stay the same. Nevertheless, it is possible that both resources map to the same value for a given point in time.

A resource can be addressed via resource identifiers, typically URIs, and whoever assigns a resource identifier is responsible to ensure the validity of the mapping semantics.

Examples could be `example.com/humans/oldest`, `example.com/humans/james-doe`, or `example.com/humans/john-doe/father`.

1.2.2 Resource Manipulation through Representations

In order to capture the state of a resource, representations are used which consist of data and metadata to describe how to interpret the data. By transferring resource representations in a REST interaction, the state of a resource can be manipulated. Typically, the data is transferred in the payload of an HTTP request or response and the metadata is transferred as HTTP headers.

Together with the HTTP verbs, such as `GET`, `POST`, `PUT` and `DELETE`, this forms a uniform interface for resource-based interactions.

1.2.3 Self-Descriptive Messages

Messages in REST interactions have to be self-descriptive, meaning that to understand a message, a component taking part in a REST interaction does not need any additional information, especially no information from previously exchanged messages.

This self-descriptiveness is achieved by including metadata in the resource representation as well as control data, such as the HTTP verb to indicate the desired action and additional HTTP headers to name the recipient of a message or control cache behavior.

1.2.4 Statelessness

Fielding writes in his thesis, "communication must be stateless in nature [...] such that 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." [Fie00, 78-79] Having self-descriptive messages is the basis for this statelessness. To achieve actual statelessness, the server then also has to be implemented in a way that it does not store session state for individual clients, instead it has to rely on the information in the self-descriptive message to process a request.

Statelessness leads to positive effects on scaling and caching. A potential drawback might be the enlarged messages sizes when repeatedly including the same data in several requests.

1.2.5 HATEOAS - Hypermedia As The Engine Of Application State

Finally, REST relies on HATEOAS. As the name already implies, hypermedia is in focus of this principle. But what does that mean - hypermedia is the engine of the state of the application?

Imagine we want to find some specific information about Alan Turing on wikipedia or any other site. First we type *https://en.wikipedia.org/wiki/Alan_Turing* into the browser to find basic information. When reading the wiki article we stumble upon the

word *enigma* and want to know more about it. So what we are naturally doing - since we are digital natives - is clicking on the hyperlink and browsing the next wiki page. Links are arrows of an interconnected graph. They navigate us to the next hypermedia (text, videos, pics etc.).

So to implement a ***hypermedia*** driven application, we can return links to subresources or other relevant resources from the requested class. This helps us to express relations between objects and enables a dynamic behavior of our application. For example, the resource representation of John Doe could include a link to his father, James Doe.

It is also possible to implement some user guidance dependent on user rights. E.g. a user gets all the possible menus he/she is allowed to use and the client dynamically renders them. This also makes changes to the API from a server perspective much easier.

Implementing HATEOAS for read-only APIs is quite easy. Just return the links to other resources and you are HATEOAS conform. One example would be the Spotify-API, presented in a short article at `api-university.com`[2]. Using operations other than read-only like `POST` is even harder since all the parameters needed for the operation must also be included in the response.

Whether to do this in the body or header is a choice of the developer. Both solutions are viable. In a first step of implementing a HATEOAS conform API, we go with the simpler approach of returning read only links.

2 Design Time

2.1 Describing your API – OAS / Swagger

Since REST is an architectural style, Fielding and his colleagues from Irvine have not published any description format/standard *how a REST API is sufficiently described!* Additionally, one can argue that because of the uniform interface principle, REST APIs can be intuitively understood. In fact, REST APIs have only been described with human-readable documentations for a long time [LG10]. Over time, however, the need for a structured interface description language (IDL) for REST APIs has been recognized. Specifying an API with an IDL enables (1) a documentation of all the **functionalities** of an API, its **paths** AND the **data formats** used in a structured, well-understood, and machine-readable way and (2) the sharing and automated processing of an interface independent of the implementation technology!

There exist a lot of ideas and IDL approaches like WADL, RAML, and API Blueprint, to name only a few, but we want to dig deeper in the world of *Open API Specification, formerly Swagger* (OAS)[3]. OAS[4] includes JSON Schema[5] for describing the input and

[2] `https://api-university.com/blog/rest-apis-with-hateoas/`

[3] Swagger Specification was open sourced in 2015 (version 2.0 of Swagger). This became Open API Specification 2.0 in 2016 (”rebranding”). A lot of tools include Swagger in their name or description, so be sensible to these two different names when searching for tools, blogs, docs etc.

[4] The current documentation can be found here: `http://spec.openapis.org/oas/v3.0.3`

[5] ”OAS uses an adapted subset of JSON Schema. It does not use all JSON Schema features, and some specific OAS features have been added to this subset”, [Lau19, 91].

output data. Lauret recommended to read first the Schema object description in OAS Specification and then the JSON Schemas official spec. OAS documents are written in YAML[6] or JSON and can be easily viewed via a plugin[7]. There are also tools to generate code for many languages, but be aware that you might get a lot of boiler plate code!

A sample OAS document looks like the following. The first line, starting with a # is a comment. Line two specifies the OAS version (important for parsers - OAS is a machine readable format), whereas the version of your implemented API is in line six. The version numbers are the only values which are surrounded by quotes.

```
#open api specification document - header with version
openapi: "3.0.4"
# general information
info:
  title: Cute Cat Service
  version: "1.0"
# reusable elements within the API specification
components:
  # data elements (JSON Schema)
  schemas:
    movie:
      type: object
      required:
        - name
      properties:
        movieId:
          type: string
        name:
          type: string
        year:
          type: number
    cat:
      type: object
      required:
        - name
        - movies
        - imageUrl
      properties:
        id:
          type: string
        name:
          type: string
        movies:
          type: array
          items:
            $ref: "#/components/schemas/movie"
        imageUrl:
          type: string
# resources (one of the REST principles)
paths:
  /cats:
    description: A catalog of all cats
    # Operations/methods via standard HTTP verbs
    get:
      summary: Get cats on a specific page
      # possible responses and their parameters
      responses:
        200:
          description: Cats on the page
          content:
            application/json:
              schema:
                type: array
                items:
                  $ref: "#/components/schemas/cat"
```

http://json-schema.org/

[6] https://yaml.org/

[7] We suggest to use IntelliJ as an IDE and therefore the recommended plug-ins would be: Swagger plugin for editing (https://plugins.jetbrains.com/plugin/8347-swagger), ReDoc plugin for viewing https://plugins.jetbrains.com/plugin/12822-redoc or an online editor like the *Swagger Editor* http://editor.swagger.io/. If you are using another IDE, find a tool set which corresponds to your needs.

The `movie` data type is specified in lines 11 – 21 and referenced within the `cat` definition in line 36. Within the `components` part there is also the possibility to add other reusable elements to your API like parameters you need. A complete example showing how to specify query and path parameters and how to deal with different response status codes can be found as a commented OAS yaml in our example project on GitHub: `https://github.com/johannes-manner/Cats`.

If you face any problems during the implementation and specification of the assignment, we highly recommend reading the OAS doc and also the JSON schema doc.

2.2 Maturity of REST APIs

So far we talked about specification and design and also about REST in general. But how can we determine whether an implemented API is REST conform (in the sense Roy Fielding defined REST)?

This is possible thanks to Richardson and his maturity model. It is a classification scheme to characterize REST APIs, also mentioned in the following papers [Pau14, RBD+16].

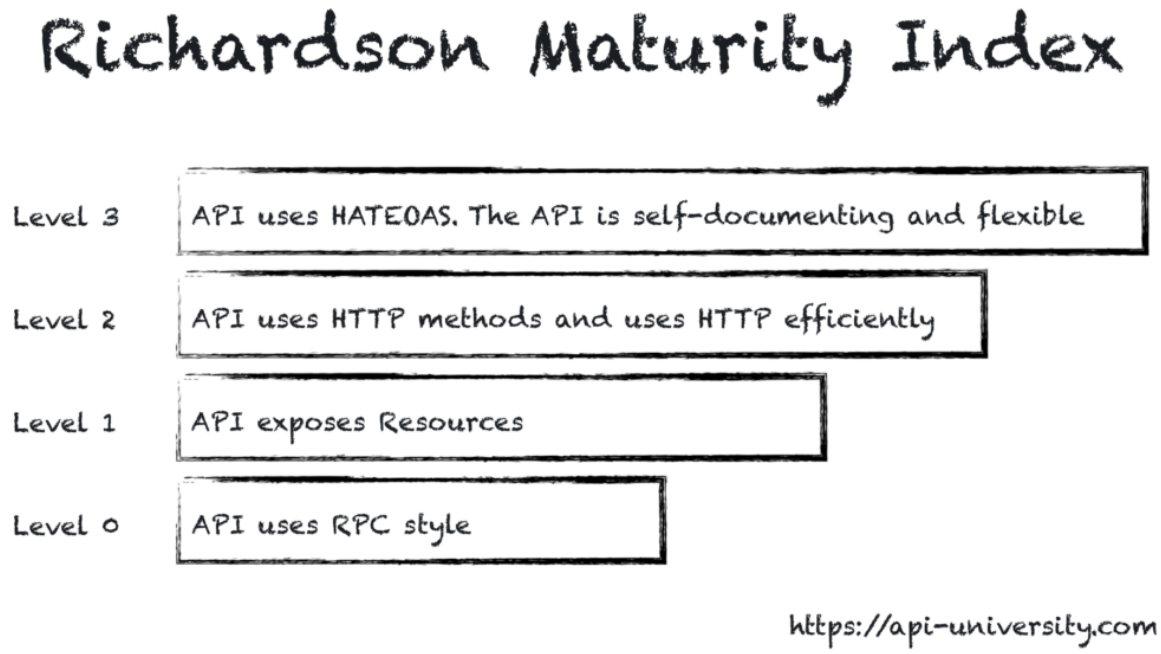

Figure 1: Richardson Maturity Model

Figure 1[8] shows the four levels of REST API maturity.

Level 0 Only using a single URI and a single HTTP verb (POST) for communication. All semantics of the interaction are included in the payload in a pre-defined format, for example relying on XML.

Level 1 Specifying a dedicated URI for every single resource. Using only a single HTTP verb (POST).

[8] `https://api-university.com/blog/richardson-maturity-model/`

Level 2 Specifying a dedicated URI for every single resource. Using all HTTP verbs to enable CRUD operations. Create (HTTP POST), Read (HTTP GET), Update (HTTP PUT) and Delete (HTTP DELETE) use corresponding status codes in the API's responses.
HINT: When people talk about REST APIs, they normally mean Level 2 APIs which are not fully REST conform.

Level 3 Also uses HATEOAS style, see Section 1.2.5 for further info. This makes the API discoverable.

Another good explanation with a running example is presented by Martin Fowler[9].

2.3 Application Architecture

The last part in this design section before we start with the implementation details deals with application architecture. The overall question is, *what does a "good" application architecture looks like when designing a REST API?*

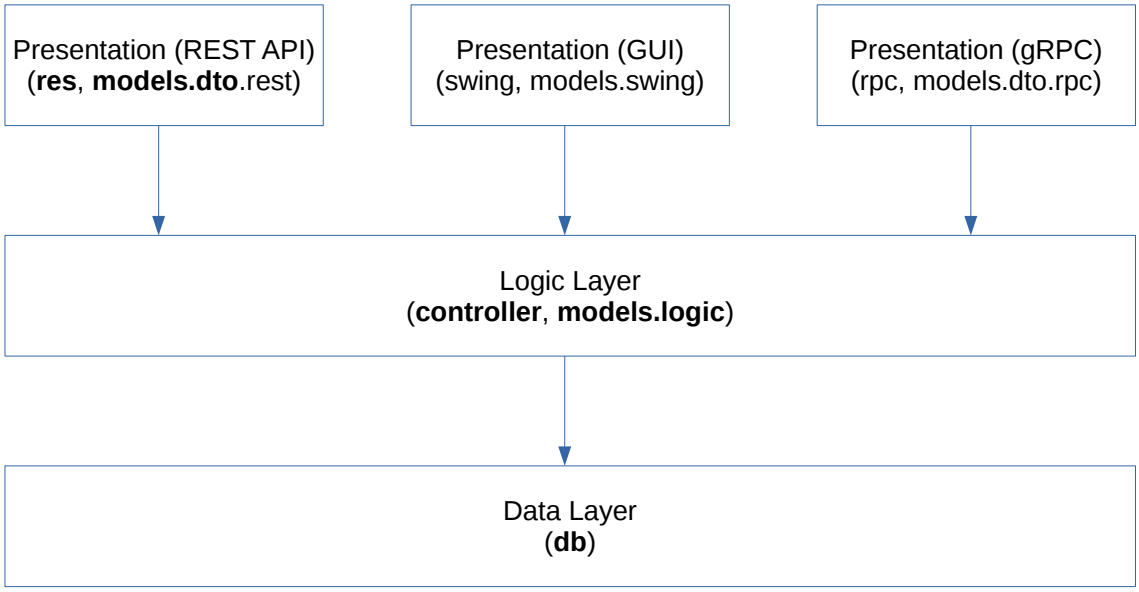

Figure 2: Layered architecture of our Cats example project

Since we are implementing an API, which is used by our customers, we want to offer services for them and hide our inner implementation. Please think carefully about this sentence. We want to hide implementation details (inner logic). Lauret writes this phrase in different forms dozent times since it is one of the core principles for designing a good API and finding a suitable architecture.

We want to recap what layered architecture means and how this could help to build a LEGO architecture.

[9] `https://martinfowler.com/articles/richardsonMaturityModel.html`

As you may know from system construction lectures and software engineering, it is a good idea to layer your system. *But why?* I agree you have to write a lot of parser methods, where you convert one format into another, but the benefit is extensibility and the possibility to exchange parts of your system without affecting others.

In Figure 2 the bold parts are packages in our example project which were implemented[10]. The arrows show the dependencies within the packages. So every layer has dependencies to the layer below or dependencies to other classes of the same layer, but no dependencies to upper layers (that is really important!!).

You also see the LEGO bricks architecture on the different representations of the presentation layer. Be aware that we have two distinct model packages, where the **dto** package is so to say a "view on the data" of the classes included in the **logic** package. So we hide our inner implementation (classes in the logic package) and decide which fields and elements we want to expose to the user (dto within the REST presentation layer).

Think about what is happening, when something changes within the logic layer, but this has no effect on the REST presentation layer? Yes, you are right - your users will not recognize the change :)

Keep this principle in mind!! (not only for this course - for your career as application developers)

3 Implementation

Now, the fun part starts. To have the full portion of implementation fun, we will not use any framework besides the Jersey reference implementation of JAX-RS.

3.1 Building a Java REST API without any Framework

Why not using a framework like Spring to build a REST API in Java? Why do we bother us with plain POJO programming in Java?

Some answers to this questions come from Marcin Piczkowski[11]. They also explain why learning REST API programming on a low level first is a good idea:

- Cleaner and more predictable code, no annotation mess.
- Control of your code, no framework magic.
- Faster startup (less dependencies).
- Smaller docker images (which is important for our second assignment).
- Foster your Java skills.

[10] The only tiny change is that we have no **res** package, we have a **resource** package.
[11] `https://dev.to/piczmar_0/framework-less-rest-api-in-java-1jbl`

If you are interested in APIs offered by big companies, tutorials and other content, search within this programmable website[12].

3.2 JAX-RS Specifications

JAX-RS (Java API for RESTful Web Services) is a Java Specification. It is described in the JSRs (Java Specification Requests) 311 (Nov. 2009 - Version 1.0), 339 (Oct. 2014 - Version 2.0) and 370 (Aug. 2017 - Version 2.1)[13]. JAX-RS is not an implementation of REST for Java, but a specification.

"Developing RESTful Web services that seamlessly support exposing your data in a variety of representation media types and abstract away the low-level details of the client-server communication is not an easy task without a good toolkit. In order to simplify development of RESTful Web services and their clients in Java, a standard and portable JAX-RS API has been designed. Jersey RESTful Web Services framework is an 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) Reference Implementation."[14]

API documentation can be found here[15].

3.2.1 JAX-RS Annotations

- **@Path('resourcePath')** – Specifies the resource path for a class or method. The paths are concatenated and result in an URI.
- **@PathParam('parameterName')** – This annotation is used in conjunction with the Path annotation. The method annotation for example is *@Path("/{id}")* and the method signature is *method(@PathParam("id") String id)*. This gives a user the opportunity to request a specific resource.
- **@QueryParam** – This annotation specifies query parameters. With *@DefaultValue ("value")* a default can assigned. The method signature is *method(@DefaultValue("red") @QueryParam("color") String carColor)*.
- **@Produces** – Specifies the MIME type for the class or a single method (overriding at method level is possible), which is produced by this method. Use the *javax.ws.rs.core. MediaType* class for implementation.
- **@Consumes** – Specifies the MIME type for the class or a single method (overriding at method level is possible), which is consumed by this method. Other MIME types result in an error. Use the *javax.ws.rs.core.MediaType* class for implementation.
- **@Context** – Necessary annotation (since JAX-RS 2.0).
  Provides for example *javax.ws.rs.core.UriInfo* to build HATEOAS links.

[12] `https://www.programmableweb.com/apis/directory`

[13] The latest JSR 370 description can be found here: `https://download.oracle.com/otndocs/jcp/jaxrs-2_1-pfd-spec/index.html`

[14] `https://eclipse-ee4j.github.io/jersey/`

[15] `https://jax-rs.github.io/apidocs/2.1/`

- HTTP verbs: **@POST**, **@PUT**, **@GET**, **@DELETE** and **@HEAD**.
- **@ApplicationPath** – Used only once as an annotation for a custom subclass of *javax.ws.rs.core.Application*. Specifies the base URI for all resources included within the application.

You can read about these annotations in more detail in an oracle tutorial[16]. There is also a running example how to use the annotations in production.

3.3 Jersey as a JAX-RS Reference Implementation

Sometimes it is easier to read parts of the documentation[17] before reading some outdated Stackoverflow or other blog post. This is only a recommendation from our side when preparing the example and the assignment.

3.3.1 Supported Media Types

JAX-RS supports mapping from POJOs annotated with JAXB[18] annotations to XML. Therefore, the supported media types from jersey are restricted as stated in the documentation in chapter 8.5.

In order to get JSON support you have to add some dependencies. Chapter 9 in the Jersey documentation describes this. We will use MOXy (preferred JSON binding support for Jersey since 2.0).

If you have the need to implement custom media types, we included an example in our Cat-Service. But be aware, that implementing custom media types is quite tricky and you leave the nice standardization way (your customers might not understand your custom media type). When you are interested, look into `provider` package and chapter 8 of the jersey documentation.

3.3.2 Content Negotiation - Self Descriptive Messages

These different media types are used for content negotiation when requesting some resources. Content negotiation is done via a specific HTTP header called `Accept`[19]. The client specifies which content types will be accepted and the server tries to fulfill this, otherwise returning a `406 Not Acceptable` HTTP status code. If the server is not capable to consume the MIME type of a client request, then JAX-RS runtime returns a `415 Unsupported Media Type` error. You can specify the provided content by the `@Produces` annotation. Keep in mind, that you can override the API wide settings when annotating specific Java function

[16] `https://docs.oracle.com/javaee/7/tutorial/jaxrs002.htm`

[17] `https://eclipse-ee4j.github.io/jersey.github.io/documentation/2.31/user-guide.html`

[18] If you are not familiar with JAXB, this article is a good starting point: `https://www.baeldung.com/jaxb` or for a more extensive discussion: `https://www.javacodegeeks.com/2014/12/jaxb-tutorial-xml-binding.html`

[19] `https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Accept`

with this annotation. We made an example where only XML can by returned when invoking `cats/search`.

3.3.3 Implementing HATEOAS - Adding Metadata to your Response

Since REST is an architectural style (no implementation), JAX-RS (also a specification - no implementation) incorporates most ideas of REST and specifies for example the MIME type via annotations to get messages self-descriptive. What JAX-RS does not support or describe is how to use hyperlinks and what is needed to be HATEOAS conform. Also the Richardson Maturity Index does not specify what is needed to be HATEOAS conform - the presence of hyperlinks is the only requirement.

Due to this lack in JAX-RS specification, a lot of frameworks arose. If you are interested in different hypermedia formats and which frameworks etc. are offered beside plain JSON, we can recommend a good comparison of HAL, JSON-LD, Collection+JSON, SIREN[20]. Since we want a fully machine readable API, we go with a combined approach between HAL and SIREN, where we structure our responses and add metadata where necesssary. The following pseudo response is in JSON format, but this is only for the sake of readability. It highlights the most important elements, especially the meta data fields and the cats part (we assume we make a GET /cats request):

```
{
  "pagination": {
    ...
  },
  "cats": {
    ...
  },
  "href": {
    ...
  }
}
```

You may recognize, that this looks really like HAL links. And you're right. In addition to the hyperlinks, we use the `OPTIONS` request to get further information about other possible operations within paths (SIREN would include this in every response, but this is not performant for bigger APIs). See our Insomnia[21] project for further details.

3.3.4 `OPTIONS` Method and its Shortcomings in Jersey

Using `OPTIONS` HTTP method on a resource, you get a WADL (Web Application Description Language) document, where all resources with their parameters are specified. This auto generated file is quite comfortable, but Jersey has its specialties when generating this WADL. There is also the possibility to access a schema of your types (XSD). This schema in combination with the operations and their in- and outputs are helpful for a explorable API, but there is also a shortcoming with this specification.

[20] `https://sookocheff.com/post/api/on-choosing-a-hypermedia-format/`
[21] `https://insomnia.rest/`

The types within XSD schema are only generated properly if the types are directly returned by the method. We use the more generic `Response` class to also specify the HTTP status code and wrap the payload via `entity` methods. A discussion about the generation problem can be found on StackOverflow[22].

Therefore, the WADL generation feature of Jersey is not usable for our purposes (not complete due to the specification we need), but the generated WADL with its resources can help to write a correct OpenAPI specification.

3.3.5 Error Messages

In many cases, it makes sense not only to return the erroneous status code, but also a user readable explanation. We decided to implement a custom class, where we categorize errors for our support team, when a user has further questions. We also include a message to resolve the problem directly.

Extend this `ErrorMessage` within your assignment and provide the user with valuable feedback and your support team in the case of an error with meaningful error types.

4 Best Practices and Hints

Best practices are *stolen* from many sources[23]

- URIs only include nouns for representations. "Singular nouns for documents, plural nouns for collections"[RBD+16, 24]. Do NOT include any HTTP verb or other operation hint into the URIs. By definition, also the version number should not be included in the URI. But it is common to use the version within the URI and we will follow this kind of API versioning[24].
- Collection of Resources. If a user requests a collection of resources, do ONLY include information on a high level basis to reduce the payload (think about the performance of your REST API).
- Single Resource. Include all the information which is stored for this resource. Also try to include a list of links from subresources which are members of this resource. The subresource information should be minimal for a powerful REST API. This

[22] `https://stackoverflow.com/questions/36212097/how-do-i-include-types-in-jersey-wadl-while-also-returning-response`
The approach is reasonable for enterprise scenarios, but we focus on writing the specification file by hand :) - It's even more fun!

[23] We tried to collect best practices when reading blogs, papers and books, these are the sources:
1 – `https://restfulapi.net/rest-api-design-tutorial-with-example/`
2 – `https://docs.oracle.com/javaee/7/tutorial/jaxrs002.htm`

[24] Other options are including it within the header, via a query parameter, creating another, new subdomain etc. These options are all not recommended. The easiest way is to design an API which is evolutionary by design and only adding optional parameters to already existing resources or adding new resources. This is not always possible, so for breaking changes use the version within the URI.

makes your REST API hypermedia driven, see the HATEOAS principle for further discussion in Section 1.2.5.

- Each resource/collection contains at least one link i.e. to itself.
- "The HTTP method POST is the default method to use when no other method fits the use case", [Lau19, 70].

Hints:

- Create operations (POST) are not idempotent since the server handles ids and uses them for creating the representation. So if you need idempotency for create operations, you have to use other mechanisms like a request ID computed out of the values of a call or something similar.
- "A @Path value isn't required to have leading or trailing slashes (/). The JAX-RS runtime parses URI path templates the same way, whether or not they have leading or trailing slashes." (2)
- "By default, the JAX-RS runtime will automatically support the methods HEAD and OPTIONS if not explicitly implemented. For HEAD, the runtime will invoke the implemented GET method, if present, and ignore the response entity, if set. For OPTIONS, the Allow response header will be set to the set of HTTP methods supported by the resource. In addition, the JAX-RS runtime will return a WADL document describing the resource; see http://www.w3.org/Submission/wadl/ for more information." (2)
- The `DELETE` HTTP method can also be used to cancel a long-running process, which was executed via `POST` but has not terminated until now. It is not always deletion as you might think from a word sense.

5 Assignment Description

DEADLINE: Sunday, 27th of June

Summer is coming and we are all looking forward to having a barbecue with friends. For a nice barbecue, we also need some refreshing drinks. Therefore, we want to start a new business where we sell beverages and deliver them to people. The easiest way is to implement a REST API where users can access the beverages via a well defined interface.

Our data model is the following:

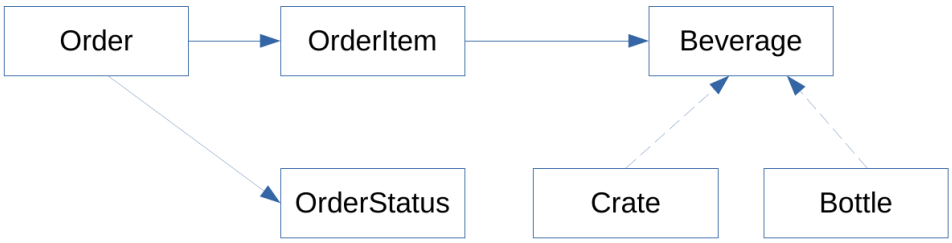

Figure 3: Model of our Beverage API

The attributes and methods can be found in the template project.

The requirements for the system are:

- Support different views on the API (we will split this into two parts in the next assignment). One view is for customers and the other one is for employees.
- A user should be able to view beverages and place orders. Implement a simple search function and a filtering mechanism with min and max price. When submitting an order, the order has the status `SUBMITTED`. The available quantity in the stock will be reduced for each beverage included in an order item. EXAMPLE: Assume there are six bottles of sparkling water in stock. If a customer wants to buy five bottles of sparkling water and submits his order, a single bottle remains in stock. So changes are made directly when submitting the order.
- To see the status and details of an order, a customer can access their order via the order's ID.
- If an order hasn't been processed yet, a customer can change their order.
- If an order hasn't been processed yet and a customer doesn't need the beverages any more, they can cancel the order. HINT: Think about the correct HTTP verb and the consequences.
- An employee should be able to create new beverages (bottles or crates), adjust their prices, quantities in stock etc. When an order is submitted by a customer, an employee has to process the orders. When the order is ok, the employee packs the beverages and commits the order as `PROCESSED`. HINT: There is no possibility for the customer to change the order any more.

- Everybody using the API should be able to access a swagger UI, deployed with our API. HINT: See the example project on GitHub to port this feature to your assignment project.

These are the minimal requirements you have to implement. They are mandatory! If you have other clever ideas how to make the API even better, contribute and implement it. (There might be some bonus points on the road :)

Hints/Remarks

- For sake of simplicity, we do not add a customer object to our application.
- Validate the incoming parameters properly (not done in the Cats demo project to keep the example simple and to see your approaches).
- We know that there is no security mechanism implemented, so malicious users could also alter the orders from other people if they have their ids (but that's not a problem for now).
- Use proper logging where possible. This helps you to find bugs and me to review your assignment.
- Add metadata where possible to make your API as explorable as possible. Implement a pagination approach.
- Design your DTOs properly (JAXB). Also think about mandatory and optional fields etc.
- Return customized error messages, which include the cause of the error and a descriptive message. Use the correct error class and suitable error status codes.

Artifacts You have to submit the following artifacts via your group's Git repository.

- Source code of your beverage store implementation.
- An Open API Specification file which fits to your implemented API.
- An Insomnia[25] project, where you "test" all your resources and methods (also with error cases you identified).

[25] `https://insomnia.rest/`

References

[Fie00] Fielding, Roy Thomas: *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[Lau19] Lauret, Arnaud: *The Design of Web APIs*. Manning, 2019.

[LG10] Lanthaler, Markus and Christian Gütl: *Towards a RESTful service ecosystem*. In *4th IEEE International Conference on Digital Ecosystems and Technologies*. IEEE, apr 2010.

[Pau14] Pautasso, Cesare: *RESTful Web Services: Principles, Patterns, Emerging Technologies*. In *Web Services Foundations*, pages 31–51. Springer New York, sep 2014.

[RBD+16] Rodríguez, Carlos, Marcos Baez, Florian Daniel, Fabio Casati, Juan Carlos Trabucco, Luigi Canali and Gianraffaele Percannella: *REST APIs: A Large-Scale Analysis of Compliance with Principles and Best Practices*. In *2016 International Conference on Web Engineering*, pages 21–39. Springer International Publishing, 2016.