Seminar paper
Marvin Westphal (3080981)
Winter term 2021/22

UNIVERSITÄT DUISBURG ESSEN

*Open-Minded*

*Seminar paper*

# Analysing the quality of generated RESTful Spring server stubs from the OpenAPI Generator

Marvin Westphal
(Matriculation number: 3080981)

Study Program: Angewandte Informatik

February 14, 2022

First reviewer: Marvin Wagner
Second reviewer: Roman Wirtz

University of Duisburg-Essen · Faculty of Engineering
CoCoS – Department of Computational and Cognitive Science
Work Group Software Engineering – Prof. Dr. M. Heisel
Supervised by: Marvin Wagner

# Contents

# 1 Introduction

"Representational State Transfer" (REST) is a paradigm for the software architecture in the client-server model. Reports from "ProgrammableWeb", the biggest repository of public web APIs with more than 24000 entries, suggest that REST has become the preferred architectural style for building web APIs. More than 80% of the registered APIs are in the style of REST (Santos, 2017). Therefore finding efficient solutions to describe and implement RESTful web services is of major significance.

The increasing usage of the REST architectural style has led to the creation of several languages to formally model and describe REST APIs. One of those languages is the OpenAPI Specification (OAS), which was founded by Swagger and is now maintained by the OpenAPI Initiative.

OAS is used to define a model of a RESTful web service. The models defined by OAS are easily readable for humans, as well as interpretable by the computer. This interpretability allows to implement supporting tools to ease the documentation, development and integration of REST APIs modelled with OAS. For example, Ed-douibi et al. (2018) propose a tool, which generates UML class diagrams, that visualize the structure and behavior of a given REST API. Furthermore, Karlsson et al. (2020) introduce "QuickREST", a tool that automatically generates property-based tests from an OAS.

The OpenAPI Generator is a tool, that allows to generate client SDKs and server stubs for several programming languages and even specific frameworks. For the generation of server stubs in the programming language Java, it offers a few generators for the framework JAX-RS in different combinations, as well as one generator for Spring (OpenAPI-Generator, 2022).

The Spring Framework is a programming and configuration model for modern Java web services. Spring Boot allows to create a standalone Spring based applications, that can be deployed easily (VMware, 2022).

This Paper aims to analyse generated Spring server stubs. For that, we will define a simplistic OAS. From that OAS onwards, we will firstly generate a Spring server stub without any configuration options set. In the second iteration we will generate a server with the option to use the delegate pattern set to true. Via the delegate pattern (Gamma et al., 1994), we will then implement the inner logic of the API.

The generated servers will then be analysed on two approaches. Firstly, we will take a look at the quality of the generated code and compare it to the Java Code Conventions (Microsystems, 1997). Secondly we will test the responses of the server stubs by providing a test module. For the server with the inner logic implemented, we will run tests for the correct logic on different requests.

# 2 RESTful web services

Fielding (2000) introduced the REST architecture style for distributed hypermedia systems. He described the architectural constraints and software engineering principles that guided the development of REST. The architecture was defined by incrementally identifying constraints that are then applied to the elements of the system. Richardson et al. (2007) built upon the initial design of REST and defined the Resource-Oriented Architecture.

## 2.1 The constraints applied to the REST architecture style

### 2.1.1 Client-server architecture

The first constraint in the REST architecture style is that it follows the client-server architectural style. This means a RESTful web service manages the communication between client and server. In this communication, a client sends a request to the server and expects a response according to that request. Thereby, the User Interface (UI) and the inner logic of the server are seperated. The components evolve independently. This improves the scalability by simplifying UI and server components. Furthermore, following the client-server model aids the architecture by improving the portability of the user interface across multiple platforms (Fielding, 2000). For example, a mobile application and a website could function on the same Back-End server logic, but be implemented in different programming languages.

### 2.1.2 Statelessness

The second constraint defines the communication between client and server as stateless. Therefore, a request to the server has to contain all of the necessary information for the server to understand and process the request. It cannot take advantage of any stored context on the server (Fielding, 2000). The Resource-oriented architecture distinguishes between application state and resource state. Application state in this context means that the client should be in charge of going through the application on its own terms. The application state is different for every client. Contrary to that, the resource state is the same for every client. The resource state is the amount of all information that can be fetched from the server (Richardson et al., 2007). If for example a picture is uploaded to be persisted on a server, it becomes part of the resource state. Following the upload, it can be fetched, updated or deleted from the server.

### 2.1.3 Cacheability

The cacheability constraint is added in order to improve network efficiency. For that, the data in the server‘s response to client‘s request should be labeled as cacheable or

noncacheable. If a response is cacheable, it can be persisted in the client´s cache temporarily . Then it has the right to be reused for later, equivalent requests. Adding a cache reduces the amount of interactions between client and server, which leads to improved efficiency and strengthens the user-perceived performance. However, if the data in the cache differs from the data that would have been requested directly from the server, the application state of the client can be incorrect. Therefore, the cache constraint can lead to reliability issues. A cache can be used on the client side, as well as the server side (Fielding, 2000).

### 2.1.4 Uniform Interface

The REST architecture style emphasizes on a uniform interface between components. That means, the information is transferred in a standardized way. This results in an overall simplified system. The visibility of interactions is improved. However, the uniform interface degrades efficiency, because the flow of information is not specifically implemented to fit a client application´s needs (Fielding, 2000). Furthermore, Fielding (2000) declares "hypermedia as the engine of application state", which means, a server provides the information to the client dynamically through hypermedia. Therefore, a generic understanding of hypermedia is all the client needs to interact with a RESTful web service.

### 2.1.5 Layered System

The constraint for a layered system enables an architecture to be built of hierarchical layers. So each component only knows about the immediate layers it is connected to. Thereby the overall system complexity is reduced. Furthermore legacy services can be encapsulated in Layers, which protects legacy clients, but allows for new services to be implemented. A layered system architecture has the weak point in it´s performance. Having the data flow through a layer of services adds overhead and latency to the processing of data. Thereby the user-perceived performance is diminished (Fielding, 2000).

## 2.2 What makes a web service RESTful?

First of all, the constraints we defined in section 2.1 have to be given for the web service. In the following, we will describe the elements in the REST architecture style and how they interact with each other.

### 2.2.1 Resource and representation

Resources and representations are the two major data elements in the REST architecture style. A resource can be any abstraction of information that can be named. The resource owns a unique resource identifier, that can be used by the client to identify the correct resource in it´s interaction with the server. The semantics of the mapping of a resource have to be static, because resources are distinguished from one another by that semantics. A resource can be made to a concept before any implementation of that concept is given, by mapping the resource to the empty set. The resource is just a conceptual mapping to a set of entities (Fielding, 2000).

The actual entity that corresponds to the mapping of the resource is the representation of that resource. The representation of a resource is the information, that is send as a series of bytes. Along with the actual data, metadata is sent to describe the bytes (Fielding, 2000). It is sent in a specific file format and language. Therefore a representation is data about the current state of a resource. Representations can flow both ways between client and server. A representation of a resource can be sent to the server to have it create a new resource or update an already existing resource. A representation of a resource can be fetched from the server, as well (Richardson et al., 2007).

### 2.2.2 Web technologies applied to REST

The REST architecture Style defines the connection between client and server. The difference between both of them is that the client initiates the connections by sending a request to the server. The server however, listens for incoming requests and responds to them according to the services, it offers. The parameters sent in a client´s request consist of a resource identifier that specifies the target resource of the request, as well as request control data and an optional representation. The parameters sent in the server´s response are response control data together with optional resource metadata and an optional representation (Fielding, 2000).

Even though the initial design of the REST architecture style does not necessarily specify a transfer protocol or a definition of the resource identifier, Richardson et al. (2007) ties REST to the Hypertext Transfer Protocol (HTTP) and the universal resource identifier (URI).

HTTP is a request/response protocol for transferring information. It allows the client to retrieve any data (e. g. plain text, hypertext, images) in quick succession, even if the servers are widely dispersed. HTTP is stateless, so it fits the constraint for statelessness in REST. For that, it establishes a TCP-connection that lasts for the duration of just one operation (Berners-Lee et al., 1994). In HTTP 1.1 Fielding and Reschke (2014) specified eight HTTP request methods. Dusseault and Snell (2010) introduces the PATCH method to the HTTP protocol. The relevant request methods to model a RESTful web service are GET, POST, PUT, DELETE and PATCH. These are described as follows:

- GET: This method requests a representation of a specified resource. It should only be used to retrieve data.
- POST: In a POST request the targeted resource processes the representation given in the request according to the resource´s own semantics.
- PUT: This method demands an update of the state of the target resource. The representation in the request payload either creates a new resource or replaces the previous state of that resource.
- DELETE: A DELETE request specifies a target resource that should be deleted from the server.
- PATCH: The PATCH method requests that a set of changes are applied to the resource found at the Request-URI. The attributes to be changed lie in the representation provided in the request.

The URI is a string that defines the address of an object on the Web. In the HTTP protocol, the URI string contains the address of the server, the client wants to contact with the request. Following this string, a substring further describes the inquired location on the server. In the URI syntax, "/" serves as a hierarchical seperator to further specify a location. The "?" seperates the address of an object to a query operation applied to that object (Berners-Lee et al., 1994). For example the URI "http://localhost:8080/books?year=1945" describes a location "books" that is located on the server "localhost:8080". The resource is further queried by the attribute "year". In a RESTful web service, the URI is used to identify resources on the server.

### 2.2.3 Example of a RESTful web service to store books

Richardson et al. (2007) describes the prototypical lifecycle of a resource as follows: A POST request creates a new resource without the client knowing the exact location. Instead the client sends a request with the representation in the entity-body to a URI, that defines the parent resource. The usual response to semantically correct POST requests is the HTTP status code 201 ("created"). The URI of the newly created resource is located in the header of that response. Once created with the client knowing the location of the resource, the resource can be modified with the PUT method, deleted with the DELETE method and a representation of that resource can be fetched using the GET method.

In the following, we will exemplify a RESTful web service of a bookstorage. From that example, we will later describe an OpenAPI Specification, that will be used to generate a Spring server stub.

The main resource of that service is the book, which contains the attributes title, author, publishing year and optionally a short summary. Furthermore it has a unique id. The POST method sends a request with an entity body that contains a representation of a book with these attributes, except the unique id. The id is created by the server. The server responds with the status created and a representation of the created book with the unique id in it. The unique id allows the client to target the resource without keeping track of the complete location. To fetch, modify or delete the resource we add the methods for GET, PUT and DELETE. In addition to the prototypical lifecycle we will add another GET method and a PATCH method. The GET method will return a list of all the books in the storage filtered by title, author or publishing year. The PATCH method will allow the client to modify single attributes of a resource, without replacing the ful resource.

# 3 OpenAPI Specification

The documentation of SmartBear (2021) describes how the OpenAPI Specification (OAS) "defines a standard, language-agnostic interface to RESTful APIs which allows both humans and computers to discover and understand the capabilities of the service without access to source code, documentation, or through network traffic inspection." Therefore, OAS aims to provide a language that can describe the model of a RESTful web service, that can be understand by humans and machines, without further knowledge of the actual web service.

A RESTful web service conforming to OAS is described in an OpenAPI document. An OpenAPI document is a JSON file, which can be represented directly in the JSON format or in the YAML format.

The OpenAPI document describes resources and the methods that can be run on these resources, as well as the different locations of these resources on the target server. Furthermore, it describes the entities used in the request- or response bodies. Section 3.1 exemplifies the modelling of an HTTP method on a resource.

## 3.1 Describing the POST method of the book storage service in an OpenAPI document

We define the model of the book storage service as an OpenAPI document in the YAML format. Listing 3.1 depicts the beginning of our OpenAPI document. In the first line we define the OAS version that is used for the specification. In the server section we define the base URL the server can by reached at. In the info section we provide further information about our API, by giving it a title, a description and adding an internal version to the API. Furthermore we add a tag and the path to the book resource.

```
1  OpenAPI: 3.0.3
2  servers:
3    - url: http://localhost:8080
4      description: Local machine
5  info:
6    title: Bookstorage API
7    description: This API stores books on a server
8    version: 0.0.1
9  tags:
10   - name: Books
11 paths:
12   /books:
```

Listing 3.1: Beginning of our OpenAPI document

First of all we define the base path of our resource as "/books". The method we describe is the POST method. For that, we firstly define the components that are involved in this method. We name the schema of the object that is sent in the entity body of the request "BookRequest". It contains the three string properties "title"", "author" and "summary", as well as the integer property "year". As we do not expect the "year" property to exceed four digits for the foreseeable future, we mark it in the smaller "int32" format. From our perspective, every book has a title, an author and a publishing year, therefore we mark these attributes as "required". To establish further understanding of the object, we add a description to the object and the underlying properties. Listing 3.2 shows the schema of the book request object.

```
components:
 schemas:
   BookRequest:
     type: object
     description: Request object of the book
     required:
       - title
       - author
       - year
     properties:
       title:
         type: string
         description: Title of the book
       author:
         type: string
         description: Name of the author of the book
       year:
         type: integer
         format: int32
         description: The year the book was published
       summary:
         type: string
         description: Short summary of the book
```

Listing 3.2: Schema of the BookRequest in the OpenAPI document

The newly created resource should then be returned in the server´s response. For that, we added the schema "BookResponse" which is similar to the schema of the book request, but has an additional, required Universally Unique Identifier (UUID) (Leach et al., 2005) "bookId". This UUID can later be used to identify the unique resource on the server. The media-types of the entity bodies are set to the JSON format. The created schemas for book request and book response are referenced in the POST method. Furthermore we need to add HTTP status codes to the response. A successful POST request should return the status code 201 ("created") (Richardson et al., 2007). For an unsuccessful request the client error status code 400 ("bad request") should be returned. This happens due to a semantical error in

the request or an invalid request syntax. To keep the API rather simplistic, we only return status code 400, even though there are several more possible status codes (e. g. 422 "unprocessable entity"), that could provide a more detailed response. We use the bad request response in all of our HTTP methods, therefore we define a reusable component for it. The POST method in the OpenAPI document is shown in listing 3.3.

```
paths:
  /books:
    post:
      tags: [Books]
      summary: Store a book
      description: Add a book to your storage.
      requestBody:
        description: Book to be created
        content:
          application/json:
            schema:
              $ref: "#/components/schemas/BookRequest"
      responses:
        201:
          description: created
          content:
            application/json:
              schema:
                $ref: "#/components/schemas/BookResponse"
        400:
          $ref: "#/components/responses/BadRequest"
```

Listing 3.3: POST method to create a book resource in the OpenAPI document

The other methods of this API were described analogue to the description of the POST method. The full API is found at https://github.com/marvinwest/openapi-generation/blob/main/bookstorage-openapi.yaml.

# 4 Approach

Now that we have a fully described OpenAPI document, we use the OpenAPI Generator to generate a Spring server stub accordingly. For that, the OpenAPI-Generator (2022) offers a command line tool, as well as a Maven plugin. To have the generated code directly in our Integrated Development Environment, we used the Maven plugin. Within the plugin, we have to define the location of the given OpenAPI document and the target locations of the generated entities and the server API. We have to describe the execution goal as "generate", as well as the generator name as "spring". Under "configOptions" we can further configure the generated server stub. Furthermore, we need to add dependencies for the Spring framework and the OpenAPI tools.

In our approach, we generate two servers with different configuration options. The first is as simple as it can be without any configuration option set (simple server). The second uses the delegate pattern (Gamma et al., 1994) and Java Optionals (delegate server). The delegate pattern of the second server will be used to implement the happy path of the inner logic of our API. The generated code will then be analysed for both servers. Furthermore, we will run a variety of tests to verify the functionality of the server stub.

## 4.1 Analysing the generated code

After the Maven project was built with the described setup, the packages for the entities and the API are present in the project´s target package. On the positive side, all entities described in the OpenAPI document are present and contain all the attributes defined in the document. Furthermore, the generated Code provides all HTTP methods. The documentation we added to these methods in the OpenAPI document is given as Javadoc above the methods. The generated code contains a class to start a Spring server with the API.

Listing 4.1 shows the beginning of the generated entity "BookRequest". From the eleven imports only five are actually used in this class. Furthermore, by using the asterisk in "import java.util.*;" all classes in the "util" package are imported, even though none of them is used in this class. Adding libraries, frameworks or other third-party software may introduce security vulnerabilities and weaknesses (Oracle, 2022). Furthermore, imports add to the code and may influence the performance of a program. Therefore, unused imports are a major flaw in the generated code. Unused imports appear in every class of the generated project.

```
1  package de.delegateserver.spring.model;
2
3  import java.net.URI;
4  import java.util.Objects;
5  import com.fasterxml.jackson.annotation.JsonProperty;
6  import com.fasterxml.jackson.annotation.JsonCreator;
7  import io.swagger.annotations.ApiModel;
8  import io.swagger.annotations.ApiModelProperty;
9  import org.OpenAPItools.jackson.nullable.JsonNullable;
10 import java.time.OffsetDateTime;
11 import javax.validation.Valid;
12 import javax.validation.constraints.*;
13
14
15 import java.util.*;
16
17 /**
18  * Request object of the book
19  */
20 @ApiModel(description = "Request object of the book")
21 @javax.annotation.Generated(value = "[..]", date = "[..]")
22 public class BookRequest   {
23   @JsonProperty("title")
24   private String title;
25
26   @JsonProperty("author")
27   private String author;
28
29   @JsonProperty("year")
30   private Integer year;
```

Listing 4.1: Excerpt of the beginning of the generated entity "BookRequest"

Listing 4.2 shows class methods on the attribute "title" in the entity "BookRequest". Even though the values for title, author and year are marked as required in the OpenAPI document, the method on top of Listing 4.2 allows to construct an object with just the title present. Therefore, constructing invalid objects is supported by the generated code. This behaviour is observed for all attributes in this class, as well as all other entities in the model package.

Both Listings show inconsistent formatting of the generated code. For example Listing 4.1 shows two blank lines in between imports. Listing 4.2 shows two blank lines between the method´s annotation and the method "getTitle" itself. Furthermore, the usage of blank lines and the maximum line length does not conform to the Java Code Conventions (Microsystems, 1997). Table 4.1 shows where the generated code does not comply to the Code Conventions.

```
1  public BookRequest title(String title) {
2      this.title = title;
3      return this;
4  }
5
6  /**
7   * Title of the book
8   * @return title
9  */
10 @ApiModelProperty(required = true, value = "Title [..]")
11 @NotNull
12
13
14 public String getTitle() {
15   return title;
16 }
17
18 public void setTitle(String title) {
19   this.title = title;
20 }
```

Listing 4.2: Excerpt of the class methods on the attribute "title" in the generated entity "BookRequest"

| Occurrence | Generated Code | Java Code Conventions |
|---|---|---|
| Blank lines between methods | partially 2 | 1 |
| Blank lines between sections | 0 - 1 | 2 |
| Maximum line length | E. g. 338 | 80 |

Table 4.1: Generated code compared to Java Code Conventions

The delegate server was generated to use Java´s Optional. This is a container object, where a value is either present or absent (Oracle, 2021). We expected it to be used on the getters of non-required attributes in the project´s entities. However, the only occurrence of Optionals is in the method header of API methods, that contain optional query parameters. Optionals are intended to be used as a method return type, where there is a need to represent an empty object (Oracle, 2021). Therefore, the OpenAPI Generator is misusing Optionals as method arguments. Furthermore, we found one occasion of commented out code without any notion on why the code is commented out.

## 4.2 Testing the generated API

We use three different procedures to test the functionality of the generated API. For that, we firstly implemented a client system, with which we can reach all the endpoints of the target server on our local machine. The client system is then used to implement the tests.

The generated server stub initially returns the HTTP status code 501 ("not implemented") for every request. We consider this to be the correct behaviour, because the server has no internal logic. Therefore, the first test procedure sends correctly formatted requests to the simple server and expects the HTTP status code 501 in every response. For this procedure, all tests passed. Therefore, every HTTP method of the simple server can be reached and responds with the expected behaviour.

In the second test procedure we want to verify, whether we can implement production code into the generated server stub. For that, we extracted the generated code of the delegate server and added it to a new Maven project. In this project, we implemented a simple service, that contains the happy path of the inner logic of the API. We used the delegate pattern to connect the service to the API, without additional expense. In this procedure we send correctly formatted requests to the delegate server and expect the correct behaviour, as described in the OpenAPI document, in the response. For example, if we add a new Book with the POST method, we expect that it can be fetched with the GET method. All tests passed for this procedure. Therefore, the inner logic of the API can be added to the generated server stub and the server responds with the expected behaviour.

The third test procedure shall verify, that the generated server responds with a client error (HTTP status code 4xx) for incorrect requests. For that, we define three types of incorrect requests. Firstly, a request that has an incorrectly formatted path parameters. Secondly, a request that is missing a required entity in it´s request body. The third type sends a request that contains the required entity in it´s request body, but that entity is missing the required attributes. For that procedure, the first two types of incorrect requests return a client error in the response. However, the third type is accepted by the server and returns a successful response (HTTP status code 2xx). For example, if we POST a book request without any of the required attributes title, author or year, the server adds the invalid object as a resource. Therefore, the generated server does not handle invalid objects correctly per default.

This behaviour can be fixed in two ways. Either, we add guards to the inner logic to verify the incoming request object or we validate the object in the API, before forwarding it to the inner logic. We would have expected the server to handle it in the second way, because all information to validate the object correctly, is given in the OpenAPI document.

# 5 Conclusion

In this paper, we described the characteristics of RESTful web services. We then showed, how to describe a model of a RESTful web service using the OpenAPI Specification. We used the resulting OpenAPI document, to automatically generate a Spring server stub using the OpenAPI Generator. The generated Spring server stub was then analysed based on the generated code and tested on it´s functionality in a production environment.

The generated code did not meet our expectations. It contained inconsistent formatting, that did not conform to the Java Code Conventions on several occasions. By adding constructors for every attribute of an entity, it allowed for invalid objects to be built. Furthermore, the generator added unused imports to every class of the project, which could lead to avoidable security and performance related issues.

Production code could easily be implemented into the generated server stub and the tests on the happy path passed. Only requests with present, but invalid objects in the request body, did not pass the tests. We expected these requests to result in a client error, but they returned a successful response. This behaviour could be fixed by updating the generated code.

The OpenAPI Generator is a valid tool to fastly generate the server stub to a RESTful web service of any size. Furthermore, by automatically generating code, the possible human error in programming is erased. This is particularly applicable to the task of copying the request and response entities described in the OpenAPI document. However, the analysis in this paper has shown, that one should not trust the OpenAPI Generator blindly. Before deploying a generated server to a production environment, the flaws described in this paper, should be revised.

The used OpenAPI document, the generated servers and the test system is found at https://github.com/marvinwest/OpenAPI-generation.

# 6 Bibliography


Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen, and Arthur Secret. The world-wide web. Commun. ACM, 37(8):76–82, aug 1994. ISSN 0001-0782. doi: 10.1145/179606.179671. URL https://doi.org/10.1145/179606.179671.

Lisa Dusseault and James M. Snell. Patch method for http. RFC, 5789:1–10, 2010.

Hamza Ed-douibi, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Openapitouml: A tool to generate uml models from openapi definitions. In Tommi Mikkonen, Ralf Klamma, and Juan Hernández, editors, Web Engineering, pages 487–491, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91662-0.

Roy T. Fielding and Julian F. Reschke. Hypertext transfer protocol (http/1.1): Semantics and content. RFC, 7231:1–101, 2014.

Roy Thomas Fielding. REST: Architectural Styles and the Design of Network-based Software Architectures. Doctoral dissertation, University of California, Irvine, 2000. URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612. URL http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1.

Stefan Karlsson, Adnan Causevic, and Daniel Sundmark. Quickrest: Property-based test generation of openapi-described restful apis. pages 131–141, 10 2020. doi: 10.1109/ICST46399.2020.00023.

P. Leach, M. Mealling, and R. Salz. Rfc 4122: A universally unique identifier (uuid) urn namespace. 2005. URL http://www.ietf.org/rfc/rfc4122.txt.

Inc. Sun Microsystems. Code Conventions for the Java Programming Language, 1997. URL http://www.oracle.com/technetwork/java/codeconventions-150003.pdf.

OpenAPI-Generator. Openapi generator, 2022. URL https://openapi-generator.tech.

Oracle. Class optional, 2021. URL https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Optional.html.

Oracle. Secure coding guidelines for java se, 2022. URL https://www.oracle.com/java/technologies/javase/seccodeguide.html.

L. Richardson, S. Ruby, and D.H. Hansson. RESTful Web Services. O'Reilly Series. O'Reilly Media, Incorporated, 2007. ISBN 9780596529260. URL https://books.google.de/books?id=RQVu5YN59loC.

Wendell Santos. Which api types and architectural styles are most used?, 2017. URL https://www.programmableweb.com/news/which-api-types-and-architectural-styles-are-most-used/research/2017/11/26.

SmartBear. Openapi specification verion 3.0.3, 2021. URL https://swagger.io/specification/.

Inc. VMware. Spring framework, 2022. URL https://spring.io/projects/spring-framework.

# Versicherung an Eides Statt

Ich versichere an Eides statt durch meine untenstehende Unterschrift,

- dass ich die vorliegende Arbeit - mit Ausnahme der Anleitung durch die Betreuer - selbstständig ohne fremde Hilfe angefertigt habe und
- dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus fremden Quellen entnommen sind, entsprechend als Zitate gekennzeichnet habe und
- dass ich ausschließlich die angegebenen Quellen (Literatur, Internetseiten, sonstige Hilfsmittel) verwendet habe und
- dass ich alle entsprechenden Angaben nach bestem Wissen und Gewissen vorgenommen habe, dass sie der Wahrheit entsprechen und dass ich nichts verschwiegen habe.

Mir ist bekannt, dass eine falsche Versicherung an Eides Statt nach §156 und nach §163 Abs. 1 des Strafgesetzbuches mit Freiheitsstrafe oder Geldstrafe bestraft wird.

| Duisburg, 14.02.2022 |  |
|---|---|
| Ort, Datum | Unterschrift |